VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Realizace funkčního vzorku koncentrátoru sériových portů

Implementation Functional Sample of Concentrator for Serial Ports

2013 Bc. Petr Havlíček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Havlíček**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Realizace funkčního vzorku koncentrátoru sériových portů
Implementation Functional Sample of Concentrator for Serial Ports

Zásady pro vypracování:

Cílem práce je realizace funkčního vzorku koncentrátoru sériových portů a jejich následné zpřístupnění prostřednictvím IP sítě. Sériový provoz z několika portů tak bude skrze enkapsulací do UDP či TCP datagramů přenášen přes lokální přepínanou síť a to plně duplexně pro každý z připojených portů.
1. Seznamte se s problematikou sériových datových přenosů nad standardem RS-232C a datových přenosů skrz lokální přepínanou síť (Ethernet).
2. Analyzujte požadavky, které na plně duplexní komunikaci vznikají, a v návaznosti na ně zvolte vhodné hardwarové prostředky pro agregaci toků ze sériových linek a jejich další přenos skrz přepínanou síť.
3. Navrhněte a zkonstruujte hardwarovou část funkčního vzorku.
4. Implementujte softwarovou část v podobě firmware pro vzniklé zařízení a jednoduchou aplikaci, které na straně PC umožní s jednotlivými datovými toky sériových portů pracovat.
5. Výsledné řešení důkladně otestujte a zhodnoťte dosažené výsledky.

Seznam doporučené odborné literatury:

AXELSON, Jan. Embedded Ethernet and Internet Complete. Madison : Lakeview Research, 2003. 482 s. ISBN 978-1931448000.
G. BLANK, Andrew. TCP/IP Foundations. Alameda, California : Sybex, 2004. 304 s. ISBN 978-0782143706.
DI JASIO, Lucio. Programming 32-bit Microcontrollers in C: Exploring the PIC32 : Embedded Technology. London : Newnes, 2008. 552 s. ISBN 978-0750687096.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Milata**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 18. dubna 2013 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli. Především své přítelkyni, která mi byla velikou oporou v průběhu tvorby této práce.

Abstrakt

Práce obsahuje analýzu technických možností pro agregaci asynchronních sériových linek a návrh takového zařízení, které je možné připojit do počítačové sítě. Pro externí komunikaci používá protokol TCP. Koncentrátor používá TCP/IP Stack firmy Microchip Technologies provozovaný na čipu řady PIC32 za použití externího řadiče Ethernetu ENC28J60. Zařízení by mělo v první řadě sloužit pro agregaci konzolových přístupů k síťovým prvkům.

Klíčová slova: tcpip, ethernet, rs232, spi, pic32, xc32, enc28j60, ansic, cisco, sítě, konzole

Abstract

The work contains an analysis of the technical possibilities for asynchronous serial port aggregation and design of this device with connection to the IP network. It uses the TCP protocol for external communication. It's based on TCP/IP Stack by Microchip Technologies running on a PIC32 series micro using external Ethernet controller ENC28J60. Equipment should be primary use to aggregate console access to network devices.

Keywords: tcpip, ethernet, rs232, spi, pic32, xc32, enc28j60, ansic, cisco, network, console

Seznam použitých zkratek a symbolů

ARP	–	Address Resolution Protocol
CD	–	Carrier Detect
CPU	–	Central Processing Unit
CR	–	Carriage Return
CSMA/CD	–	Carrier Sense Multiple Access with Collision Detection
CTS	–	Clear To Send
DHCP	–	Dynamic Host Configuration Protocol
DMA	–	Direct Memory Access
DNS	–	Domain Name System
DSR	–	Data Set Ready
DTR	–	Data Terminal Ready
EEPROM	–	Electrically Erasable Programmable Read-Only Memory
EIA	–	Electronic Industries Association
EUI	–	Extended Unique Identifier
FIFO	–	First In First Out
FPGA	–	Field-Programmable Gate Array
FTP	–	File Transfer Protocol
GND	–	Ground
HTTP	–	Hypertext Transfer Protocol
HW	–	Hardware
IBM	–	International Business Machines
ICANN	–	Internet Corporation for Assigned Names and Numbers
ICD	–	In-Circuit Debugger
ICMP	–	Internet Control Message Protocol
ICSP	–	In Circuit Serial Programming
IDE	–	Integrated Development Environment
IEEE	–	Institute of Electrical and Electronics Engineers
IETF	–	Internet Engineering Task Force
IP	–	Internet Protocol
IPv4	–	Internet Protocol Version 4
IPv6	–	Internet Protocol Version 6
ISO	–	International Organization for Standardization
ITU	–	International Telecommunication Union

LAN	–	Local Area Network
LED	–	Light-Emitting Diode
LF	–	Line Feed
LLC	–	Logical Link Control
MAC	–	Media Access Control
MCU	–	Microcontroller Unit
MISO	–	Master In Slave Out
MOSI	–	Master Out Slave In
NAT	–	Network Address Translation
OS	–	Operating System
OSI	–	Open Systems Interconnection
OUI	–	Organizationally Unique identifier
PC	–	Personal Computer
PCB	–	Printed Circuit Board
PCI	–	Peripheral Component Interconnect
PDU	–	Protocol Data Unit
RFC	–	Request For Comment
RI	–	Ring Indicator
RISC	–	Reduced Instruction Set Computing
RM	–	Reference Model
RS	–	Recommended standard
RTS	–	Request To Send
RU	–	Rack Unit
RxD	–	Received Data
SCK	–	Serial Clock
SMD	–	Surface Mount Device
SNTP	–	Simple Network Time Protocol
SoC	–	System on Chip
SOIC	–	Small-Outline Integrated Circuit
SOT	–	Small-Outline Transistor
SPI	–	Serial Peripheral Interface
SS	–	Slave Select
SSH	–	Secure Shell
SSOP	–	Shrink Small-Outline Package
SW	–	Software
TCP	–	Transmission Control Protocol
TQFP	–	Thin Quad Flat Package
TTL	–	Time to Live
TxD	–	Transmitted Data

USB	–	Universal Serial Bus
UTP	–	Unshielded Twisted Pair
VCP	–	Virtual COM Port
VHDL	–	VHSIC Hardware Description Language
VHSIC	–	Very-High-Speed Integrated Circuit
WAN	–	Wide Area Network

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Za poslední roky již klasický sériový port prakticky vymizel ze segmentu osobních počítačů a notebooků. Mnoho zařízení jej ovšem stále používá pro komunikaci s vnějším světem. Jeho oblíbenost na straně výrobců je dána především jednoduchostí obsluhy komunikace. Absence těchto portů na PC se často řeší pomocí zařízení připojitelných do USB vytvářejících virtuální sériový port. Výrobci tento postup používají i při rozšíření svých produktů o USB konektivitu. Pouze na straně zařízení přidají čip obstarávající převod USB na UART rozhraní.

V posledních letech se čím dál více objevuje trend připojit všechna zařízení do počítačové sítě. Jeden příklad za všechny může být sdílení USB zařízení přes počítačovou síť, kdy stačí zařízení připojit do chytrého USB rozbočovače a poté si z kteréhokoliv počítače na světě připojit zařízení pouhým kliknutím. S tímto trendem souvisí i potřeba připojit starší periferie do počítačové sítě. Velmi lákavá je představa systému, kde by stačilo připojit jakékoliv zařízení se sériovým protokolem a hned by bylo dostupné na síti.

Tvorba takového zařízení není zrovna nejjednodušší. Je potřeba zvážit mnoho kroků od původního nápadu až po výsledný produkt. Tato práce je průvodce, jak tak učinit, včetně popisu problémů, se kterými na začátku počítáte či nikoliv. Je to lehce trnitá cesta, kde je potřeba mít znalosti z oblastí programování, sítí, elektroniky a strojírenství. Snažil jsem se při návrhu zúročit poznatky získané za dobu studia a současné metody návrhu zařízení.

Výsledkem je modulární zařízení použitelné jako platforma pro převod sériových (synchronních i asynchronních) datových přenosů na protokol TCP používaný v počítačových sítích. Vše je otázkou návrhu koncového modulu zajišťujícího převod mezi koncovým protokolem a interní SPI sběrnicí.

Zařízení je postaveno za pomoci technologií firmy Microchip Technologies, protože s nimi mám velmi dobré zkušenosti. V oblasti TCP/IP sítí tato firma nabízí komplexní ověřené řešení. Licenční politika je také velmi příznivá. Pro bezplatné použití i v komerčních aplikacích je pouze nutné provozovat dané řešení nad MCU téže firmy. Stejnou platformu používá mnoho vývojářů na celém světě, takže v případě problému není těžké sehnat podporu.

2 Motivace

Sériové rozhraní se velmi uchytilo v prostředí správy aktivních síťových prvků, které sice umožňují vzdálenou správu pomocí protokolů telnet a SSH, může ovšem nastat situace, že tato správa není při prvním spuštění povolena. Prvky mají rozhraní pro lokální správu, jenž se označuje jako konzolový port. Jedná se o port používající asynchronní sériový přenos s normou RS-232. Velmi často také může dojít k znepřístupnění vzdálené správy z důvodu chyby v síti nebo chybou správce. V takovém okamžiku bývá jediná možnost administrace pomocí konzolového portu. Na straně PC se pro komunikaci s tímto rozhraním aktivního prvku používá tzv. COM port. Tento název se vznikl podle původní funkce tohoto portu, sloužil totiž primárně k připojení komunikačního zařízení (modemu). Používá asynchronní sériový přenos standardu RS-232.

Síťové prvky se často sdružují do rozvaděčových skříní. Důležité síťové uzly jsou většinou umisťovány do specializovaných sálů s klimatizací a zajištěním bezvýpadkového napájení. V takových sálech se může vyskytnout velké množství prvků, pro které je nutné mít možnost správy i v případě problému v síti. Nebude již tedy stačit server se dvěma COM porty, ale je nutné mít specializovaný HW disponující desítkami takovýchto portů. Nejčastěji volená varianta je koupě PCI karty, ke které se připojí specializovaný kabel zakončený zpravidla 8 COM porty. Toto řešení je plně funkční, ale je velmi nákladné vzhledem k počtu takto získaných portů. Navíc musí neustále běžet server obsahující danou rozšiřující kartu a počet rozšiřujících slotů je velmi omezen.

Velká agregace síťových prvků nastává také v případě laboratoří počítačových sítí. V běžné laboratoři ale není potřeba centrální správa všech prvků, protože každý student konfiguruje pouze malý počet prvků a stačí mu sériový port osobního počítače. Diametrálně odlišná situace nastává v případě virtuální laboratoře. Zde bývá přístup do správy jednotlivých prvků zprostředkován skrze centrální prvek (konzolový server). Na katedře informatiky VŠB-TU Ostrava vznikla takováto virtuální laboratoř s názvem Virtlab. Tato práce má ulehčit možnosti rozšíření této laboratoře o další síťové prvky bez nákladů spojených s připojením prvků ke konzolovému serveru.

Pro úsporu času u složitějších úkolů je při inicializaci úlohy do jednotlivých prvků nahrána konfigurace. Nahrání těchto konfigurací zajišťuje konzolový server. Pro zrychlení celého procesu se používá paralelní přístup na více prvků najednou.

Z výše uvedeného textu jasně vyplývají požadavky na agregační zařízení. Mělo by se jednat o zařízení umístitelné do datového rozvaděče. Musí obsahovat velké množství (více než 10) sériových portů. Ideálně bude propojeno s konzolovým serverem pomocí TCP/IP sítě. Předpokládá se plně duplexní režim při paralelním přístupu na více portů. Pro správnou funkci je také důležitá odezva celého systému pro přistupujícího uživatele. Je nutné také počítat s konfigurací prováděnou konzolovým serverem, jenž je v porovnání s uživatelem mnohem rychlejší. Musí být zajištěno spolehlivé spojení mezi konzolovým serverem a portem pro správu zvoleného prvku. Nesmí docházet ke ztrátě dat nebo k doručení dat na nesprávné koncové rozhraní.

3 Komunikační protokoly

Digitální přenos si lze představit jako sériový tok bitů po médiu. Pro úspěšný přenos dat mezi komunikujícími uzly je nutné, aby všichni účastníci věděli, jaká data jsou přenášena. Hlavně je nutné zajistit konzistentní sémantiku dat na obou koncích komunikace. Pro splnění této podmínky jsou definovány a standardizovány komunikační protokoly. Specifikace jednotlivých protokolů jsou velmi komplexní. Neobsahují jen popis reprezentace jednotlivých bitů, ale také například časování mezi bity, možná přenosová média a další parametry nutné pro správnou funkci.

Firmy většinou nezveřejňují specifikace vlastnoručně vytvořených protokolů. Nezřídka se tato uzavřenost využívá pro zamezení použití produktů jiných výrobců v jejich systému. Zákazník je poté nucen koupit veškeré součásti od jednoho výrobce. Další možný přístup k této problematice je povolení použití protokolu na základě licence. Součástí licence může být finanční poplatek za využívání protokolu. Tento model používají i konsorcia starající se o tvorbu několika velmi známých standardů (například USB, Zigbee), kde jsou poplatky využity pro financování vývoje nových specifikací.

Existují také otevřené standardy. Vznikající pod záštitou nadnárodních institucí. Mezi nejznámější organizace patří EIA, ISO, IEEE, IETF a ITU. V oboru počítačových sítí se často setkáte s dokumenty RFC, což jsou veřejně dostupné dokumenty vydávané IETF. Z principu nejde o standard jako takový, ale pouze o doporučení. Nicméně se jich velká část výrobců drží pro zajištění interoperability s jinými výrobci.

3.1 TCP/IP

Mezi takové patří také TCP/IP. Jde o rodinu protokolů označovanou jako Internetová. Zkratka je tvořena z názvů dvou nejpoužívanějších protokolů této rodiny a to TCP a IP [1]. K jednoduchému určení závislostí mezi protokoly existuje několik hierarchických modelů. Tyto modely seskupují protokoly do úrovní. Data poté putují od nejvyšší vrstvy do nejnižší a z každé vrstvy musí použít právě jeden protokol. Nejznámější je bezpochyby model organizace ISO s názvem RM OSI definovaný v [2] zobrazený na obrázku 1. Model je sice určen pro popis OSI sítí, ale dá se velmi dobře aplikovat i na TCP/IP sítě. Ne všechna zařízení musí mít implementované všechny vrstvy modelu. Počet nutných vrstev je závislý na funkci zařízení, ale vždy musí jít o souvislé vrstvy od nejnižší směrem nahoru. Například přepínač potřebuje pro svou funkci mít implementovanou spojovou vrstvu. Jak se má dostat k jednotlivým rámcům, když neumí extrahovat jednotlivé rámce z media? Proto musí mít implementovanou i první vrstvu. Stejnou logiku lze aplikovat i na zařízení pracující na vyšších vrstvách.

3.1.1 Fyzická vrstva

Řadí se zde především standardy definující vlastnosti signálu a jejich přenosu po médiu. Samozřejmě jsou zde definovaná samotná přenosová média, jež lze rozdělit do tří základních skupin:

- metalická,

Obrázek 1: Hierarchický model RM OSI

- optická,
- rádiová.

Všechny 3 jsou v počítačových sítích velmi časté. Uživatelé počítače se ovšem nejčastěji setkají s metalickým vedením. Pro přenos dat v TCP/IP sítích se používá kroucená nestíněná dvoulinka známá pod zkratkou UTP. Podle přenosových parametrů se jednotlivé UTP dělí do kategorií. Pro rychlejší přenosy je nutné použít vyšší kategorie kabelů. V současnosti se běžně setkáváme s rychlostí 100 Mb/s pro koncové uživatele a 1000 Mb/s pro servery. Obě tyto rychlosti se dají přenášet po kategorii 5e. Pro 1 Gb/s se ale doporučuje použít kategorii 6 z důvodu lepší odolnosti proti zarušení.

Na této vrstvě jsou už data přímo ve formě jednotlivých bitů. Řeší se tu jejich reprezentace na médiu. Od jejich kódování přes časování přenosu až po přenosové úrovně na médiu. Patří sem například DSL a ISDN. Do této vrstvy zasahuje i Ethernet, kde je definice přenosu pro jednotlivá média.

3.1.2 Spojová vrstva

Spojová vrstva má 2 podvrstvy. Blíže síťové vrstvě je LLC zajišťující zapouzdření paketů do rámců. Přidává nejen záhlaví, ale i zápatí obsahující kontrolní součet pro ověření správnosti doručení. Pod ní je pak vrstva MAC zprostředkovávající přístup k médiu. Nejčastěji se používá přístupová metoda CSMA/CD. Ta přikazuje vysílající stanici před začátkem přenosu naslouchat na médiu, zda již nekomunikuje někdo jiný. Pokud ano, musí počkat na uvolnění média. Když je médium prázdné, může začít vysílat. Když dva nebo více uzlů budou chtít komunikovat ve stejný okamžik, dojde ke kolizi. První, kdo zjistí existenci kolize, začne vysílat předem definovaný signál informující ostatní o vzniku kolize. Všichni, kteří chtějí vysílat, čekají náhodnou dobu a po jejím uplynutí znova naslouchají na médiu. Tento problém se netýká spojení s plně duplexním přenosem.

Nejznámějším zástupcem spojové vrstvy je Ethernet, který se postupem času stal vlastně jediným protokolem pro LAN. Využívá fyzické adresování pomocí MAC adres. Každý síťový adaptér má při výrobě přidělenou globálně unikátní adresu standardu EUI-48. Adresa má 48 bitů a dělí se na dvě stejně velké části. První 3 bajty jsou známy jako OUI a identifikuje výrobce adaptéru. Výrobce může zakoupit skupinu adres se stejným OUI od organizace ICANN. Zbylé bajty si výrobce přiděluje dle svého uvážení dané kartě. V záhlaví protokolu Ethernet je umístěna zdrojová a cílová MAC adresa, platná pouze v rámci jednoho IP úseku.

Při komunikaci zná počítač IP adresu cíle, ale pro přenos potřebuje znát také MAC adresu. Pro její získaní byl vytvořen protokol ARP [3]. Ten zajistí mapování IP adresy na MAC adresu. Funkce je velmi jednoduchá. Počítač pošle dotaz na všechna zařízení v IP segmentu, kdo má hledanou IP adresu. Stanice s danou adresou odpoví svou MAC adresou. Tento krok je potřeba před každou komunikací. Pro urychlení přenosu a snížení zatížení sítě se výsledky ARP protokolu uchovávají v cache. Nejprve se prohledá cache, až když není daný záznam nalezen, přistupuje na řadu ARP.

3.1.3 Síťová vrstva

Dnes nejpoužívanější protokol síťové vrstvy je IP. Hlavní funkcí IP je logické adresování uzlu sítě a zprostředkování komunikace mezi uzly v různých sítích, ale provádí také zapouzdření segmentů do paketů přidáním IP hlavičky. V Internetu je nejvíce používán IP ve verzi 4. Rychle ho ovšem dohání IP verze 6 [4]. Hlavní nevýhodou verze 4 je malý adresový prostor. Pro adresování se používá 32bitová IP adresa zapisovaná jako 4 číslice 0-255 oddělené tečkami. V době vzniku byl celkový počet adres 2^{32} takřka nekonečný. Velmi brzo po rozšíření Internetu jsme se ale k hranici maximálnímu počtu adres přiblížili. Následkem čehož došlo k vyčerpání IPv4 adres. Problém dočasně vyřešila technologie NAT umožnující za jednu veřejnou adresu skrýt více privátních. Ovšem ani ta již není dostačující. Konečným řešením má být plné nasazení IPv6 s 128bitovou adresou zapisující se již v hexadecimálním tvaru s dvojtečkou jako oddělovačem. Prostor se zatím taky zdá nekonečný, avšak je jen otázka času, než opět dojde k jeho vyčerpání. Nasazení IPv6 velmi komplikuje množství změn proti IPv4. Nejde totiž jen o zvětšení délky adresy, ale o předefinování základních principů fungování IP, které se navíc pořád dál a dál rozšiřuje.

Je definováno několik typů komunikace. Přenos probíhající mezi pouze dvěma uzly se nazývá unicast. Datový tok přijímající více cílů se nazývá multicast. Používá se často pro distribuci video-obsahu více stanicím v síti. Poslední variantou je broadcast, u kterého přijímají všechny uzly v dané síti. Multicast má definovanou adresu, jež se chová stejně jako broadcast. V IPv6 už není podporován broadcast a plně ho nahradil multicast. Novinka je anycast, při kterém existuje více zařízení se stejnou IPv6 adresou a provoz je vždy směrován na nejbližší cíl s danou adresou.

Adresa není kompletní bez síťové masky. Maska má stejnou délku jako adresa a je to zleva spojitá řada jedniček. V místě přechodu z jedničky na nuly dělí adresu na 2 části. První určuje adresu sítě. Druhá je určena pro hosta. Jsou dvě adresy, které nesmí host použít. První má v části pro hosta samé nuly, je označována jako adresa sítě a používá se při směrování pro identifikaci celé sítě. Druhá má v části pro hosta samé jedničky a je

to adresa pro broadcast pro danou síť. Stanice jsou schopny komunikovat přímo v rámci jedné sítě. Když je cíl mimo aktuální IP segment, tak se provoz posílá na adresu výchozí brány. Ta by již měla vědět, co se zadaným provozem udělat, pokud neví, provoz se zahodí. Výchozí brána je směrovač spojující lokální segment se zbytkem sítě.

IP má integrovanou ochranu proti nekonečnému zacyklení paketů. Při průchodu směrovačem pokaždé dochází k dekrementování položky TTL v IP hlavičce. V případě IPv6 se pole jmenuje "Hop Count". Při dosáhnutí nuly je paket zahozen a směrovač pošle odesílateli ICMP zprávu "Time Exceeded". Této vlastnosti vyžívá mapovací nástroj traceroute, který postupným inkrementováním TTL získává IP adresy (zdrojové adresy v ICMP zprávách) jednotlivých směrovačů mezi zdrojem a cílem.

IP protokol používá jako svůj řídící protokol ICMP [5]. ICMP pracuje nad IP, ale přitom je součástí jeho implementace, takže patří do stejné vrstvy. Má několik zpráv, z nichž je nejznámější *echo request* a *echo reply*, používaných programem ping. ICMP informuje například o zahození z důvodu vypršení TTL nebo nedostupnosti cílové adresy.

3.1.4 Transportní vrstva

Účel této vrstvy je rozlišení služeb a programů aplikační vrstvy. Rozlišení se provádí pomocí portů. Každý přenos má zdrojový a cílový port (z dynamického rozsahu). Porty mají 16 bitů a podle [6] se dělí do těchto kategorií:

- 0-1023 - dobře známé,
- 1024-49151 - registrované,
- 49152-65535 - dynamické.

Provádí zapouzdření dat získaných z vyšší vrstvy do segmentů. Děje se tak přidáním hlavičky protokolu TCP nebo UDP k datům. Jednodušší z nich je UDP [7]. Není totiž spojově orientovaný a nezaručuje doručení dat. Nemusí tak řešit navazování spojení a potvrzování přijetí dat. Ve velmi zobecněném pohledu jen posílá data a nezajímá se, zda je někdo přijme. Otázkou je, zda má takový protokol smysl. K čemu je dobré používat protokol, když nevím, jestli poslaná data někdo vůbec přijme? Nesmíme ovšem zapomenout, že mluvíme o protokolu 4. vrstvy. Tudíž kontrolu doručení může zajistit například vyšší vrstva. Své největší uplatnění má v aplikacích požadujících velmi rychlý přenos s malým zatížením sítě. Příkladem takových přenosů je video a hlas, kdy je nutné, aby byl přenos rychlý a když se něco ztrácí, tak je lepší použít úspěšně přenesená data než čekat na opětovné doručení ztracených kousků.

TCP [8] musí před začátkem přenosu sestavit spojení a to nadále udržovat. K přenosu používá 2 čísla umístěná v hlavičce. Jmenují se sekvenční (Sequence) a potvrzovací (Acknowledgment) číslo. Jejich inicializace probíhá během sestavování spojení. Slouží pro zajištění spolehlivého přenosu a správného pořadí. TCP má řízení toku. To zajišťuje tzv. okno. Velikost okna je umístěno v TCP hlavičce a mění se v průběhu komunikace. Okno je počet odeslaných bajtů než je doručeno potvrzení o přijetí. V případě, že je nastaveno na nulu, dojde k pozastavení komunikace. Obvykle se velikost okna řídí aktuálním stavem

vstupního bufferu. Nemůže tak dojít k přehlcení a ke ztrátě dat. Tato vlastnost je velmi výhodná v případě realizace koncového zařízení na MCU, kde jsou paměťové možnosti velmi omezené. Po dokončení přenosu je nutné spojení ukončit.

3.1.5 Aplikační vrstva

Nejvyšší vrstva modelu obsahuje protokoly rozdělené do dvou skupin. První obsahuje uživatelsky používané protokoly například HTTP, FTP a SSH. Druhou skupinu tvoří podpůrné protokoly, starající se o správnou funkci sítě. Zde lze zařadit DNS, DHCP, SNMP atd. Při tvorbě agregátoru budou důležité hlavně podpůrné protokoly.

Pro automatickou konfiguraci koncových stanic se používá protokol DHCP [9], používající komunikaci server-klient. Za pomoci 4 základních zpráv je schopen stanici nakonfigurovat základní síťové parametry. Ve skutečnosti toho však umí daleko více. Veškeré informace a nastavení se přenáší pomocí tzv. options, kde každá má svůj unikátní 8bitový identifikátor. V těchto položkách se běžně ukrývá IP adresa pro stanici, síťová maska, adresa výchozí brány a DNS serverů. Adresu dostává stanice vždy na určitou dobu, po které je nutné si o adresu znova zažádat, pokud k tomu nedojde, je uvolněna pro další použití. Je dobrým zvykem zažádat si o adresu v půlce doby zapůjčení. Doba zapůjčení je také přenášená jako atribut při DHCP komunikaci. Při připojení zařízení do sítě pošle zprávu DHCP Discovery s cílovou adresou 255.255.255.255, což znamená globální broadcast. Nachází-li se DHCP server ve stejné podsíti, tak odpoví zprávou DHCP Offer, kde uvede síťové atributy. Stanice ovšem nemůže tyto parametry použít, dokud nedojde k potvrzení ze strany serveru. Může nastat situace, kdy bude v jednom segmentu sítě několik DHCP serverů a všechny servery se musí dozvědět, kterou odpověď serveru si klient zvolil. Žádost o získané údaje ze strany klienta probíhá zprávou DHCP Request. Poslední kousek skládanky je zpráva DHCP Acknowledgement sloužící jako potvrzení od serveru. Pro komunikaci se musí použít protokol UDP, protože nejde sestavit spojení bez zdrojové IP adresy.

3.2 RS-232C

Sériová komunikace mezi PC a periferiemi byla po dlouhou dobu realizovaná pomocí rozhraní označené COM. Šlo o asynchronní sériové rozhraní používající standard EIA RS-232C. Tento standard se řadí do fyzické vrstvy. Definuje zapojení konektorů, elektrické charakteristiky a časování při komunikaci. Rozhraní COM se objevovalo ve dvou provedeních a to 9 a 25kolíkový lichoběžníkový konektor typu canon. Pokud mělo PC oba konektory, tak 9kolíkový byl COM1 a 25kolíkový COM2. V současné generaci PC se tyto konektory objevují jen zřídka. Častěji výrobci pouze umístí na desku rozšiřující konektor reprezentovaný dvouřadou kolíkovou lištou. Za posledních pár let bylo rozhraní nahrazeno pomocí USB. Z důvodu ušetření místa na panelech koncových zařízení se lze také setkat s konektory RJ-45, které nemají piny RI a CD. Pro přenos jsou nejdůležitější signály pro příjem (RxD), vysílání (TxD) a společnou zem (GND).

Norma původně vznikla k umožnění připojení modemu k PC, proto jsou zde signály bez užitku v dnešní době. Jsou to signály RI a CD sloužící k vytáčení a příjímání spo-

jení. RS-232 disponuje řízením toku. Koncové uzly jsou schopné signalizovat, zda jsou připraveny přijímat data. Používá se kombinace DTR a DSR nebo RTS a CTS. Zapojují se do kříže. DTR pin jednoho uzlu je spojen s DSR druhého. Stejně to funguje i pro druhou kombinaci. V prostředí správy aktivních prvků se nepoužívá.

Norma definuje napěťové úrovně používané při komunikaci na tomto rozhraní. Všechny napěťové hladiny mají společnou zem. Je nutné ji vždy zapojit, jinak komunikace nebude fungovat. V případě, že zařízení jsou na jiných napájecích okruzích, může nastat situace, kdy zem obou zařízení nemá stejný potenciál. Následkem takového stavu dojde k průtoku proudu přes společnou zem v datovém kabelu. To přináší rušení a zahřívání části v cestě proudu. Pro přenos se používá symetrické (kladné i záporné) napětí. K přenosu logické 1 se používá vždy záporné napětí. Nejčastější napěťové úrovně jsou +15/-15 V a +12/-12 V. Záleží na dostupnosti napětí pro koncový budič. Běžně se používá obvod MAX232 původně navrhnut firmou Maxim, jenž má v sobě integrované 2 nábojové pumpy. První zajišťuje zdvojnásobení napětí, druhá se stará o invertování tohoto napětí. Obvod má asymetrické napájení 5 V. Dělají se i varianty s jiným napájením, ale ty již nejsou tak běžné. Zajímavé je, že standard nedefinuje maximální délku kabelu. Namísto toho definuje maximální kapacitu vedení mezi konci. Běžné kabely lze použít do délky 15 m, což je 3x více, než umožňuje norma pro USB.

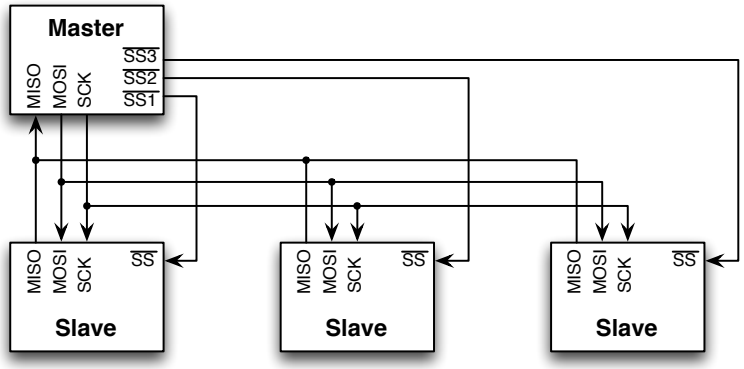
Komunikace začíná vysláním start bitu. Poté následuje předem daný počet datových bitů. Většinou se používá 8, ale můžeme potkat i 9 bitů. Bit navíc se používá pro rozlišení mezi daty a příkazy. Po přenesení dat následuje předem nastavený počet stop bitů. Kontrola konzistence dat se dá realizovat přes paritní bit. Nastavení musí být na obou koncích stejné. U zařízení připojitelných na COM port lze tyto údaje najít v dokumentaci. Nejběžněji se objevuje označení 9600 8n1, což lze rozkódovat jako:

- rychlost 9600 baudů,
- 8 datových bitů,
- bez parity,
- 1 stop bit.

3.3 SPI

Pro rychlou komunikaci mezi periferiemi a mikrořadiči byla společností Motorola vytvořena sběrnice SPI /citespi. Velkou předností je plně-duplexní přenos dat. Neumožňuje však jednostrannou komunikaci. Strana, která pouze přijímá, musí vysílat nějaká data (běžně samé nuly). Datový provoz navíc může být taktovaný až na desítky megahertz. Uzly sběrnice se dělí na dva typy. První je Master, jenž řídí komunikaci na sběrnici. Druhý je Slave, který sám o sobě nemůže komunikovat. SPI podporuje několik Slave i Master zařízení na jedné sběrnici. Více Master zařízení se příliš nepoužívá, proto zde rozvedu topologii s jedním Master a více Slave uzly zobrazenou na obrázku 2.

Signál SCK slouží pro přenos hodinového signálu od Master zařízení k Slave uzlům. Na straně Slave uzlů není možnost ovlivnit hodinový signál. Pomocí signálu MOSI dochází k přenosu dat z Master na Slave. Naopak MISO přenáší data z Slave do Master.



Obrázek 2: Topologie SPI sběrnice

Aktivní Slave se volí pomocí separátního pinu $\overline{SS}$. Výběr se provádí přivedením logické nuly na $\overline{SS}$ příslušného Slave obvodu.

SPI podporuje 4 módy časování. Liší se polaritou a fází. U polarity je na výběr, zda je klidová úroveň logická nula nebo jednička. V případě fáze záleží na hraně, při které dochází ke vzorkovaní dat. Na výběr je vzestupná a sestupná hrana. Konfigurace časování se může měnit při přepínání mezi Slave zařízeními tak, aby splnila specifikace daného výrobce zařízení. Lze tedy kombinovat obvody od několika výrobců s různou maximální rychlostí a s různou konfigurací hodinového signálu.

Komunikaci vždy zahajuje Master nastavením pinu $\overline{SS}$ do logické nuly. Posléze se při každé vzestupné nebo sestupné hraně hodinového signálu umístí na MOSI (zapisuje Master) a MISO (zapisuje Slave) jeden bit dat. Přenos je realizován pomocí posuvného registru. Pro ukončení přenosu stačí přepnout $\overline{SS}$ do logické jedničky. Potřebuje-li Slave komunikovat, musí dát vědět uzlu Master, který pak zahájí komunikaci.

Z fyzického hlediska musí být piny účastnící se komunikace po SPI třístavové. Během trvání logické jedničky na $\overline{SS}$ jsou piny SCK, MOSI, MISO ve stavu vysoké impedance. Na sběrnici se nenalézají žádné pull-up odpory ani jiné součástky.

4 Možnosti realizace

Každý problém má několik řešení. Uvedu zde několik metod, jak zadaný problém řešit. Všechny realizace mají společné základní rysy. Musí jít o autonomní zařízení připojitelné do počítačové sítě LAN. Musí umožňovat paralelní přístup na více portů. Nesmí docházet ke ztrátě dat z jednotlivých portů ani v jednom směru komunikace. Výsledné zařízení musí být vyrobitelné. Programovací jazyk C je velkou výhodou není však podmínkou. Pro koncové porty počítám s použitím minimálního zapojení bez podpory řízení toku. Půjde tedy jen o piny RxD, TxD a GND.

Hlavním problémem je vytvoření dostatku koncových portů. Není problém vytvořit větší počet výstupních budičů kompatibilních s RS-232C. Horší je zajistit řídicí logiku k těmto obvodům. Tato problematika se nám komplikuje o obsluhu počítačové sítě, které zajisté spotřebuje velké množství strojových cyklů. Z toho vyplývá, že nejde použít pouze jeden CPU. Zůstává možnost rozdistribuovat úlohy mezi více obvodů. Dobrou volbou je využít řadič Ethernetu, se kterým odpadne starost o linkovou vrstvu. Dělají se také obvody s plně implementovaným TCP/IP stackem, ale ty nejsou zrovna nejlevnější. Jsou určeny pro rozšíření již existujících obvodů o propojení do sítě.

Nejvhodnější je realizovat sériovou komunikaci pomocí hardware modulu vytvořeného k tomuto účelu. Hlavní výhodou je automatické přijímání celého bajtu na příslušné rychlosti. Odpadá tak starost s hlídáním časovače a rotací registrů. Další velkou výhodou je vyvolání přerušení při přijetí dat. Není tedy nutné periodicky kontrolovat vstupní buffer, zda v něm nejsou nezpracovaná data. Nespornou výhodou tohoto řešení je také odlehčení CPU s obsluhou. Stačí pouze po přečtení dat oznámit CPU, aby je zpracoval. Tyto moduly se běžně vyskytují integrované v mikrořadičích. Problém nastává s jejich počtem. Většina obvodů disponuje pouze jedním až dvěma. Maximum, se kterým jsem setkal, bylo 4. Toto číslo není dostačující, tak je potřeba poohlédnout se po jiném řešení, jak tento počet zvýšit.

4.1 Software UART

Při čistě SW implementaci musí CPU řešit celé řízení přenosu. Hned na začátku narazíme na několik problémů. Prvním je časování. Při příjmu i odesílání je potřeba dodržet časy, ve kterých se data čtou a posílají. Bez použití jakékoliv HW podpory máme možnost pouze čekat daný čas v nějaké smyčce. CPU mezitím nemůže dělat nic jiného než čekat. Je jasné, že nemůže probíhat několik paralelních přenosů najednou. Dokonce ani nejde paralelně přijímat a vysílat najednou. Teoreticky by mělo jít počkat s vysíláním až na začátek příjmu. Sice pak bude potřeba hlídat interval jen jednou, ale komunikace selže, když nebude dlouhodobě co přijímat. Další velký problém je detekce začátku přenosu. Bez HW podpory je nutné periodicky kontrolovat RxD pin, zda náhodou nezačal přenos.

Jednoduše by se dalo říct, že striktně SW implementace není optimální cesta jak agregovat více UART modulů do jednoho obvodu. Existuje však možnost, jak pomocí dobře dostupných HW modulů polo-automatizovat příjem dat.

Pro ulehčení procesoru je možné využít běžně dostupných HW modulů, které jsou daleko častější než samotné UART moduly. Pro udržení konstantní rychlosti příjmu je

možné použít časovač. Protože jde o asynchronní komunikaci, musí být pro každý SW UART vlastní časovač. Komunikaci může začít kdokoliv a kdykoliv. Stačí mu poslat start bit a poslat data. K odchycení začátku komunikace musí být pin pro čtení opatřen přerušením. Po vyvolání přerušení se teprve spustí časovač a začne se generovat přerušení v době platných stavů na lince.

Ve výsledku jsme problém trochu zjednodušili, ale pořád není ani zdaleka vyřešen. Nyní je pro realizaci jednoho UART modulu nutné mít 2 časovače a 1 zdroj externího přerušení. Tyto prostředky jsou sice častější, ale pořád bude možnost vytvořit maximálně 8 UART modulů. A to jsou již velmi drahé čipy s velkým počtem vývodů, které zde nebudou využity. Navíc nezbude dostatek zdrojů pro ostatní funkcionalitu zařízení.

4.2 Hradlové pole

Programovatelná hradlová pole FPGA jsou velmi univerzální součástkou, která obsahuje v zobecněném případě pouze základní logické struktury, ze kterých lze vytvořit složitější obvody. Je to nástroj umožňující provádět hardwarovou implementaci složitých obvodů, které nejsou závislé na chodu procesoru. Touto metodou lze interně vytvořit několik sériových modulů. Stejnou metodu je možné použít i na implementaci autonomního Ethernetového řadiče. Výsledek této implementace se označuje jako SoC. Běžně by byla potřeba několika samostatných integrovaných obvodů pro realizaci. FPGA umožňuje sjednotit funkci všech těchto obvodů do jednoho a interně je propojit sběrnicí.

Při implementaci takového návrhu uvnitř obvodu FPGA je k dispozici několik úrovní nástrojů. Nejnižší je přímo programování celého řešení v jazyce VHDL. Tato metoda je velmi pracná a dala by se přirovnat k psaní strojového kódu pro procesor. Další možností návrhu je použít již hotové bloky označované jako Intellectual property. Jsou to vlastně logické celky plnící danou funkčnost. Některé se dají sehnat i pod otevřenou licencí a je možné je použít pro soukromé i komerční aplikace. Mezi takovými bloky není problém najít například USB, PCI, Ethernet. Jsou k dispozici i bloky realizující funkci celého procesoru; nejznámější je MicroBlaze vytvořený přímo firmou Xilinx.

Toto řešení má také své nedostatky. Při návrhu je velmi jednoduché udělat chybu, která se velmi složitě hledá. Není zde k dispozici debugger, jak ho známe z klasického programování. Pro ladění se používá častěji logická sonda. Z pohledu návrhu PCB je nutné zohlednit, že je doporučené použít minimálně 4vrstvou desku se samostatnými vrstvami pro napájení. Navíc je celkem snadné způsobit nestabilitu a zakmitání FPGA, když člověk nemá zkušenost ohledně návrhu s těmito obvody. Nikdy jsem navíc nedělal kompletní návrh s použitím těchto obvodů. Vždy šlo pouze o psaní kódu v VHDL a poté jeho provoz na vývojové desce vytvořené výrobcem.

4.3 Modulární systém

Chtěl jsem mít implementaci založenou na MCU. Mám s nimi letitou zkušenost v oblasti SW i HW návrhu. Velké množství MCU má v současné době k dispozici kompilátor pro ANSI C a C++. To umožní tvorbu přehledného kódu, který se v budoucnu bude dát snadno rozšířit.

Řešení s jedním hlavním procesorem měla jednu skrytou vadu. Nebylo je možno rozšířit o jiné typy portu. Primárně sice má být agregátor určen k připojení konzolových portů síťových prvků, ale chci mít možnost připojit i jiné protokoly. To by nebylo v případě jediného čipu možné. V případě potřeby použít například komunikace za pomocí RS-422 by se musela dělat celá nová deska s novým budičem.

Celý koncept je již dobře známý a je používán u robustních řešení v oblasti automatizace a řízení. Jde o to mít řízení oddělené od koncových prvků. V mém případě jde o oddělení řídícího čipu a koncových portů. Bude tedy jeden MCU jako řídící a pak MCU pro každý koncový port. Jejich propojení se bude realizovat za pomoci standardního komunikačního protokolu. Koncový modul tak může vytvořit kdokoliv, kdo bude mít k dispozici popis této interní komunikace. Výsledkem této úvahy je vytvoření univerzální platformy pro převod sériového toku dat na TCP/IP. Řídicí obvod vlastně jen přenáší data z interní sběrnice do TCP soketu a obráceně. Koncový modul zase obstarává výměnu dat z UART modulu na interní sběrnici.

Interní komunikace samozřejmě neprobíhá paralelně se všemi koncovými moduly najednou. Pokud ale bude probíhat sériově a dostatečně rychle, tak se bude jevit jako paralelní. V době, kdy modul data neposílá, je ovšem nutné zajistit, aby data byla umístěna do vyrovnávací paměti. Doba aktivní komunikace musí být dostatečně dlouhá na přenos všech dat umístěných v bufferu. Jinak by v případě velkého datového toku hrozilo přehlcení vyrovnávací paměti, což by mělo za následek ztrátu dat.

Algoritmus použitý pro přepínaní portu je znám jako Round-robin. Porty se nepreemptivně přepínají jeden po druhém. Dobu přepínaní lze volit dvěma způsoby. Jeden je přepínat porty po konstantních časových úsecích. Tato metoda minimalizuje jitter v komunikaci. Ve výsledku bude tedy konstantní zpoždění mezi klientem na TCP soketu a koncovým zařízením. Další výhodou je ochrana proti selhání jednoho portu. Nemůže dojít k tomu, že jeden port bude chtít přenést více dat, než může za daný úsek stihnout a tak ovlivnit ostatní koncové porty. Při této metodě dochází ke stavům, kdy sběrnice nepřenáší data. Druhá metoda eliminuje tato prázdná místa v komunikaci. Po přenesení dat pro daný port dochází okamžitě k přepnutí na další port. Ve výsledku to výrazně sníží zpoždění, protože málokdy probíhá nepřetržitý provoz. Pro ochranu před uvíznutím je nutná externí kontrola počtu dat k přenosu, aby nebyla větší než přenositelné maximum.

Pro rozumnou propustnost je vhodné mít interní sběrnici s podporou plně duplexní komunikace, neboť i většina koncových portů je plně duplexní. Je vhodné použít protokol s HW podporou v MCU. Navíc musí jít o dobře známý protokol, protože modul musí být nejen v hlavním MCU, ale také ve všech koncových MCU. Z těchto důvodů je ideální volbou SPI.

5 Finální řešení

S ohledem na výsledné umístění zařízení jsem zvolil krabici určenou k montáži do datového rozvaděče. Tyto krabice mají na čelní straně standardizovaný úchyt. Šířka rozvaděče je dána na 19 palců, ale výška a hloubka je proměnná. Výška se měří v RU, kde 1 RU má hodnotu 44,45 mm. Krabice se vyrábějí v celých násobcích RU. Zařízení není interně prostorově náročné, proto jsem zvolil nejmenší možnou výšku 1 RU. Hloubka byla dána aktuálním stavem trhu. Hledal jsem dostupnou variantu s co nejmenší hloubkou. Původně jsem chtěl krabici z kovu. Postupem času jsem však tento názor změnil a pořídil nakonec variantu z ABS s označením G17081UBK výrobce Pro Power. Má hloubku pouhých 200 mm, což je o 10 cm méně než kovový ekvivalent. Navíc se ABS opracovává snadněji než kov. Třetinová pořizovací cena (přibližně 300 Kč) oproti kovové byla také jeden z důvodů mé volby. Má ale i své zápory. Krabička se spojuje šrouby z vrchní strany, to přináší nutnost sloupků uvnitř krabičky pro vedení šroubů. Plast je navíc křehčí a je tedy nutné opatrné zacházení.

Pro udržení strukturální integrity krabice musel výrobce použít silnější stěny. Tato vlastnost se ukázala jako výhoda a vyřešila problém s uchycením hlavního plošného spoje a zdroje. Nechtěl jsem žádné výčnělky, které by mohly překážet při instalaci několika zařízení nad sebou. Díky širší stěně dna krabičky nebyl problém použít šrouby M3 se zápustnou hlavou. Zdroj má otvory přímo se závitem M3. Pro uchycení hlavní PCB byla použita sestava zápustného šroubu M3 o délce 6 mm, kovového distančního sloupku výšky 8 mm se závity z obou stran a šroubu M3 s délkou 6 mm. Zápustný šroub je skrze spodek krabičky vlepen do distančního sloupku speciálním lepidlem. Tento krok zajistí ochranu proti povolení a tedy protáčení distančního sloupku. Z vrchní strany je již deska klasicky přišroubována. Důležité je dodržet délky šroubu, jinak nebude možno oba šrouby plně dotáhnout.

Další náročnou částí konstrukce bylo umístění desek koncových modulů. Těchto PCB se do zařízení musí vejít celkem 16 (každá ponese 2 koncové porty). Problém je v tom, že všechny koncové porty musí být na předním panelu. První koncept byl jednoduchý. Šlo o to vyfrézovat do vrchního a spodního krytu drážku dlouhou stejně jako PCB. Deska plošného spoje je větší než vnitřní výška krabice, takže se bude pohybovat jen v této drážce. Při realizaci bohužel nastala jistá komplikace ohledně přesnosti výroby. Sebemenší odchylka mezi vrchní a spodní drážkou má za následek naklonění celé desky. Ve výsledku pak konektory nepasovaly do čelního panelu. Bohužel jsem tento problém neočekával a zjistil jsem ho až při osazení desek do krabice. Tento stav inicializoval nový nápad na uchycení. PCB bude fixováno v oblasti konektoru pomocí předního panelu. Na druhém konci pak bude spoj umístěn do malé drážky vyřezané v kovové tyči 12 x 5 mm s hloubkou 5 mm. Tato tyč bude umístěna z obou stran. Veškeré úpravy jsem prováděl vlastnoručně.

Hlavním zdrojem napájení byl zvolen produkt firmy Mean Well s označením RS-15-5. Jde o spínaný zdroj disponující výkonem 15 W při nominálním výstupním napětí 5 V. Výstupní napětí se dá nastavit od 4,75 V do 5,5 V. To umožňuje dát na vstup desky diodu chránící před přepólováním, která vždy způsobí úbytek napětí. Správným nastavením zdroje pak lze dosáhnout napětí 5,00 V na výstupu diody. Zdroj je napájen

přímo z rozvodné sítě 230 V/50 Hz. Připojení je realizováno přes konektor standardu C14 s integrovaným pouzdrem pro tavnou pojistku. Pouzdro je osazeno pomalou pojistkou T 400 mA/250 V. Mezi konektorem a zdrojem je zapojen ještě dvoupólový vypínač s doutnavkou, jenž je umístěn na čelním panelu. Vodiče pro síťové napětí mají z bezpečnostních důvodů dvojitou izolaci. Vidlice a vypínač jsou propojeny konektory fast-on s izolační bužírkou. Zdroj má pouze svorkovnici se šroubky pro umístění kabelových oček. K usnadnění montáže jsem použil izolované vidlice pro nalisování na kabel.

Návrh schémat a desek plošných spojů probíhal v programu EAGLE [11] firmy Cadsoft ve verzích 6.2.0 až 6.4.0. Program je řádně zakoupen s licencí i pro komerční použití. Licence omezila maximální velikost desky na 160x100 mm, proto jsem nemohl navrhnout propojovací desku mezi koncovými moduly a hlavní deskou. Toto úskalí jsem vyřešil použitím dvouřadých konektorů pro ploché kabely. Výrobu výsledných návrhů plošných spojů jsem přenechal firmě PragoBoard s.r.o. formou POOL servisu. Při tomto postupu se účtují pouze poplatky podle velikosti desky a neúčtují se náklady na výrobní podklady. Tento postup je velmi výhodný při výrobě prototypů. Nevýhodou je omezení možností zvolit si fyzické provedení, jako například síla měděné vrstvy, barva nepájivé masky nebo barva servisního potisku. Firma touto formou vyrábí PCB s 2 i 4 vrstvami. Samozřejmost jsou prokovené vrtané otvory.

Fyzická konstrukce je uzpůsobena pro 16 koncových modulů. Celkem tedy jde o 32 koncových portů. Výsledek této práce ovšem bude mít pouze 4 moduly. Je to dostačující množství na ukázku funkčnosti tohoto návrhu. Tento počet není zvolen náhodou. Výrobce PCB má minimální velikost při účtování zakázky stanovenou na 1 dm^2. Spojil jsem proto 4 desky koncového modulu do jednoho PCB. Ten byl lehce větší než udaná minimální velikost. Nedošlo tak ke zbytečnému přeplacení desky.

Zdrojové kódy jsou psány v programovacím jazyce C s využitím kompilátoru XC32 (v1.20) a XC8 (v1.12) firmy Microchip Technology Inc. Oba kompilátory byly použity ve volné verzi, což znamená absenci optimalizace výsledného kódu. Pro vývoj jsem použil prostředí MPLAB X IDE v1.60 vytvořený stejnou firmou. Vývoj probíhal v operačním systému Mac OS X 10.8, ale projekt jde otevřít na kterémkoliv OS, kde lze provozovat MPLAB X. Vývojové prostředí je postaveno na platformě Netbeans. Pro dodatečné nástroje, jako například TCP/IP Stack, jsem použil Microchip Libraries for Applications v2012-10-15. Jedná se o balík knihoven a ukázkových kódů pro práci s nimi. K nahrání programu do MCU je nutný hardwarový programátor. Při tvorbě této práce jsem použil ICD3 stejné firmy, který má přímé napojení na IDE. ICD3 je zároveň i debugger s podporou breakpointů a krokování programu. Programování probíhá skrze sériový protokol, proto lze programovat již osazené SMD součástky.

5.1 Externí komunikace

Pro komunikaci s vnějším světem má hlavní PCB k dispozici dvě rozhraní. První je dobře známé USB. Použil jsem čip společnosti Future Technology Devices International Ltd. s označením FT232RL. Tato firma se specializuje na tvorbu elektronických součástek s podporou USB. Dříve zmíněný obvod se v principu chová jako most mezi USB a UART. Není to zcela nejmodernější model této firmy, ale mám s ním dlouholetou zkušenost.

Podporuje všechny obvyklé operační systémy, na OS Linux je dokonce ovladač již přímo v kernelu a to od verze 3.0.0-19. Obvod je napájen ze strany počítače pomocí USB kabelu. Původní úmysl byl vytvořit úspornější zařízení. Proč by měl být napájen USB řadič, když se bez kabelu stejně nedá používat? Má to ale jeden vedlejší efekt. K usnadnění ladění jsem využil dvou pinů pro umístění LED indikující aktivitu na USB. Jednotlivé piny s LED jsou ovšem korektně inicializovány až při připojení napětí k obvodu. Zjednodušeně řečeno, po restartu zařízení budou obě LED svítit do prvního připojení kabelu. Zapojení je převzato z katalogového listu obvodu. Používá modul UART2 hlavního MCU. Hlavní důvod této volby je překrytí pinů modulů UART1 a SPI1. Nezapojil jsem vývody určené k řízení toku při komunikaci. Není žádný praktický důvod takto činit. Protože jsem nemusel příliš řešit místo, použil jsem konektor USB B klasické velikosti.

Po připojení USB kabelu dojde k vytvoření virtuálního COM portu (za pomoci VCP ovladače). Pro práci s ním lze použít libovolnou terminálovou aplikaci. Konfigurace sériového přenosu je 9600 8N1 bez řízení toku. Po zapnutí dojde k výpisu základních informací o agregátoru. Součástí výpisu je zobrazení počtu nastavených portů a TCP porty pro přístup k nim. Během inicializace TCP/IP Stacku je vypsána IP adresa pro připojení. Pokud není uložena žádná předchozí adresa použije se výchozí linková IP adresa 169.254.1.1/24. Po stisku klávesy 'c' přejde konzole do konfiguračního módu zobrazeného na výpisu 1. Zde jde ručně modifikovat důležité parametry aplikace. Konfigurace se ovládá pomocí číselných voleb a jde o konfiguračního průvodce. Aktuální konfiguraci lze zapsat do EEPROM. Pro ukončení lze vystoupit pouze z průvodce, nebo restartovat celé zařízení. Možnost restartu je zde pro případ změny TCP portu nebo jejich počtu, aby proběhla korektní inicializace soketů. Dodal jsem nastavení počtu portů a počátečního TCP portu, jenž bude použit pro řídící soket. Jde o předělanou verzi výpisu dodaného v ukázkovém příkladě v souboru UARTConfig.c.

```
Configuration wizard

MAC Address:                        00:04:a3:3b:c2:ea

    1: Change host name:                AGGREGATOR
    2: Change static IP address:        10.0.0.31
    3: Change static gateway address:   10.0.0.1
    4: Change static subnet mask:       255.255.255.0
    5: Change static primary DNS server:  10.0.0.1
    6: Change static secondary DNS server: 8.8.8.8
    7: Disable DHCP & IP Gleaning:      DHCP is currently enabled
    8: Change Port Count:               8
    9: Change Starting TCP Port:        1500
    w: Save.
    q: Quit.
    r: Reboot.

Enter a menu choice:
```

Výpis 1: Úkázka konfiguračního režimu

Další možnost komunikace je skrz IP síť. Model komunikace je zřejmý. Každý koncový port má přiřazen svůj TCP port. Cokoliv se pošle na daný TCP port se objeví na

příslušném koncovém portu. Stejné pravidlo platí i pro opačný směr komunikace. Využil jsem integrovaný obvod ENC28J60, který v sobě integruje kompletní Ethernet. Schází mu jediná věc, a to je MAC adresa. Ta má být celosvětově unikátní. Sehnat jednu adresu od ICANN není zrovna nejjednodušší úkol, proto se v tomto typu projektů používá adresa existujícího avšak nefunkčního síťového adaptéru, u kterého nemůže dojít k opětovnému připojení do počítačové sítě. Zvolil jsem trochu jiný přístup. Protože jsem potřeboval ukládat nastavení do energeticky nezávislé pamětí, kterou zvolený MCU nemá integrovanou, jsem musel tak použít externí paměť. Po usilovném hledání jsem nalezl paměť EEPROM komunikující na SPI, která řeší oba problémy. Jedná se o integrovaný obvod 25AA02E48 vyrobený společností Microchip Technology. Označení E48 udává, že se jedná o paměť s předkonfigurovanou unikátní EUI-48 adresou. Přesněji to je adresa s OUI 00:04:A3. Velikost paměti je 256 B. Není to sice mnoho, ale je to dostačující na ukládání stavu TCP/IP. Adresa začíná na adrese 0xFA.

Podpora EEPROM paměti je přímo integrovaná v TCP/IP Stacku, ale výchozí implementace má několik chyb znemožňujících komunikaci se zvoleným modelem. První z problémů je s zápis adresy. Většinou se posílá adresa o velikosti 16 nebo 24 b. 25AA02E48 potřebuje zaslat 8 b adresu. Další je s dávkovým zápisem. Ten se provádí po stránkách. Předkonfigurovaná velikost stránky je 64 b. Zvolená paměť podporuje velikost maximálně 16 b. Musel jsem upravit obsluhu SPI EEPROM, aby podporovala zvolenou paměť. Aktuální konfigurace se ukládá ve struktuře *APP_CONFIG* definované v souboru StackTsk.h. Při ukládání aktuální konfigurace se ukládá právě tato struktura. Na výpisu 2 jde vidět, že není nikterak rozsáhlá. Využil jsem již existujících funkcí pro práci s touto strukturou k uchování dodatečných nastavení mé aplikace. Stačilo pouze dodat potřebné proměnné do struktury a o zbytek se postarají již hotové funkce.

```
typedef struct __attribute__ ((__packed__)) appConfigStruct
{
    IP_ADDR    MyIPAddr;                // IP address
    IP_ADDR    MyMask;                  // Subnet mask
    IP_ADDR    MyGateway;               // Default Gateway
    IP_ADDR    PrimaryDNSServer;        // Primary DNS Server
    IP_ADDR    SecondaryDNSServer;      // Secondary DNS Server
    IP_ADDR    DefaultIPAddr;           // Default IP address
    IP_ADDR    DefaultMask;             // Default subnet mask
    BYTE       NetBIOSName[16];         // NetBIOS name
    BYTE       PortsCount;              // Count of End Port
    WORD       MgmtTCPPort;             // TCP Port for managment socket
    struct
    {
        unsigned char : 6;
        unsigned char bIsDHCPEnabled : 1;
        unsigned char bInConfigMode : 1;
    } Flags;                            // Flag structure
    MAC_ADDR MyMACAddr;                 // Application MAC address

} APP_CONFIG;
```

Výpis 2: Konfigurační struktura

Pro konfiguraci je součástí balíku i generátor konfigurace. Je to uživatelsky velmi přívětivá grafická aplikace postupně provázející konfigurací celé aplikace. Výsledkem je soubor TCPIPConfig.h obsahující všechny náležitosti k úspěšné funkci. I přes práci, jakou musel někdo strávit při tvorbě této aplikace, jsem zvolil manuální konfiguraci přímo v souboru. Nebyl to velký problém z důvodu velmi dobře komentovaného obsahu. Definují se zde služby a protokoly, které mají být povoleny. Výpis 3 ukazuje, že jsem použil pouze základní protokoly nutné pro správnou funkci. DHCP je pro vývojáře implementované velmi jednoduše. Stačí odkomentovat definici *STACK_USE_DHCP_CLIENT* a zařízení již v případě dostupnosti DHCP serveru získá adresu samo. Nabídka protokolu je opravdu velmi rozsáhlá a je vhodná snad pro jakýkoliv hardware připojený do počítačové sítě. Implementovaná je třeba podpora pro FTP nebo HTTPS s ověřením uživatele. Zajímavá je implementace klienta a serveru pro testování propustnosti sítě za pomoci nástroje *iperf*. Protokoly jako SMTP jsou pak pouze bonusem dokazujícím robustnost celého řešení. V souboru jsou dále definovány výchozí parametry pro nastavení sítě včetně MAC adresy a IP adres. Výchozí nastavení se přebírá pouze v případě, kdy není k nalezení uložená konfigurace.

```
#define STACK_USE_UART              // UART for IP address display and stack configuration
#define STACK_USE_ICMP_SERVER       // Ping query and response capability
#define STACK_USE_ICMP_CLIENT       // Ping transmission capability
#define STACK_USE_DHCP_CLIENT       // Dynamic Host Configuration Protocol client
#define STACK_USE_DNS               // Domain Name Service Client
```

Výpis 3: Konfigurace povolených služeb TCP/IP

Nejdůležitější sekce je konfigurace TCP. Každé aktivní spojení potřebuje ke své existenci úložiště pro uložení svých informací. Navíc každý soket má svou vstupní a výstupní vyrovnávací paměť. Interně se realizuje přes FIFO buffer. Zde začíná počítání s pamětí. Velikost RAM je omezená, tak je nutné si dobře rozmyslet kolik můžeme dát každému soketu. Paměť je rozdělena do několika částí. Každá se konfiguruje zvlášť, jak jde vidět na výpisu 4. První je RAM uvnitř Ethernet řadiče, kde je k dispozici 8 kB. Při prvotních pokusech se mi nepodařilo tuto možnost dobře využít, proto jsem se rozhodl pro jiné řešení. Další blok, kde se dají ukládat data soketů, je interní RAM MCU. Tu jsem zvolil jako primární a ukládám zde veškerá data. Poslední možnost je použít externí RAM připojenou přes SPI. V ní lze dokonce určit, kterou část přesně použít pomocí počáteční adresy. Zvolil jsem hodnotu 16 kB, která je plně dostačující i pro 32 koncových portů.

```
#define TCP_ETH_RAM_SIZE            (0ul)
#define TCP_PIC_RAM_SIZE            (16000ul)
#define TCP_SPI_RAM_SIZE            (0ul)
#define TCP_SPI_RAM_BASE_ADDRESS    (0x00)
```

Výpis 4: Konfigurace paměti TCP

Pro rozlišení soketů mezi sebou se definují typy. Definice je velmi snadná, jak ukazuje výpis 5. Jde vlastně jen o zpřehlednění kódu, klidně lze bez negativního vlivu přímo psát hodnoty 0, 1 atd. Jen výsledný kód bude nečitelný. V této aplikaci potřebuji jen 2 typy soketů. Jeden slouží pro propojení s koncovými porty. Šlo by jej charakterizovat jako

datový soket. Druhý je řídící soket určený pro vzdálený přístup. Je to klon lokálního USB. Funguje na stejném principu, jenom se spouští připojením klienta.

```
#define TCP_SOCKET_TYPES
    #define TCP_PURPOSE_END_PORT        0
    #define TCP_PURPOSE_MANAGMENT       1
#define END_OF_TCP_SOCKET_TYPES
```

Výpis 5: Definice typů soketů

Poslední část nastavení je určení maximálního počtu soketů pro jednotlivé typy, umístění v paměti a velikost bufferů. Celá operace se provádí přes zajímavou strukturu ukázanou na výpisu 6. Zobrazená definice je pro maximálně 8 koncových portů a 1 řídící soket. Každý koncový port má v RAM rezervovaných 250 B pro vstupní a 250 B pro výstupní data. Je velmi důležité zvolit správné hodnoty, jsou totiž použity pro určení velikosti okna během TCP komunikace. Díky této funkci nenastane případ, kdy by došlo k přetečení vstupního bufferu.

```
#define TCP_CONFIGURATION
    ROM struct
    {
        BYTE vSocketPurpose;
        BYTE vMemoryMedium;
        WORD wTXBufferSize;
        WORD wRXBufferSize;
    } TCPSocketInitializer[] =
    {
        {TCP_PURPOSE_MANAGMENT, TCP_PIC_RAM, 200, 200},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
        {TCP_PURPOSE_END_PORT, TCP_PIC_RAM, 250, 250},
    };
#define END_OF_TCP_CONFIGURATION
```

Výpis 6: Inicializační struktura

Zbývá provést konfiguraci protokolu UDP. Ta je oproti TCP dost zjednodušená, jak ukazuje výpis 7. Je nutné pouze definovat maximální počet souběžných soketů, které má zařízení umět zpracovat. Z důvodu zvýšení propustnosti je vhodné vypnout počítání kontrolního součtu u odchozích UDP datagramů. Rozdíl výkonnosti může být až 50 %. Toto omezení se týká hlavně použitého řadiče. Obvody řady PIC32MX6xx/7xx a ENCx24J600 mají rychlý DMA přístup do paměti zrychlující tento proces.

```
#define MAX_UDP_SOCKETS (4u)
// #define UDP_USE_TX_CHECKSUM
```

Výpis 7: Konfigurace UDP

Vše výše uvedené se týkalo konfigurace. Na výpisu 7 je ukázán začátek inicializace celé aplikace. Jako první je vždy nutné inicializovat hardwarové moduly. Tuto činnost zajišťuje funkce *hw_init*, kde dochází k inicializaci hardwaru MCU, jako třeba UART a SPI modulu. Další funkce načte předprogramovanou EUI-48 adresu z paměti EEPROM. Teď máme vše potřebné pro inicializaci TCP/IP. První volaná funkce inicializuje HW prostředky pro TCP/IP, některé používá pro svou práci i UART. Následující funkce *InitAppConfig* zajistí naplnění konfigurační struktury výchozími hodnotami, pokud se podaří nalézt uloženou konfiguraci v EEPROM, je načtena uložená. Musel jsem do ní dopsat inicializaci mnou přidaných položek ve struktuře. Vkládám do nich hodnoty zadané pomocí maker na začátku souboru. Ty po prvním uložení ztrácí význam, protože se vždy načte uložená konfigurace. Poslední funkce provádí inicializaci všech potřebných protokolů.

```
hw_init () ;

XEEReadArray(0xfa, SerializedMACAddress, 6);

 TickInit () ;
InitAppConfig() ;
StackInit () ;
```

Výpis 8: Úvod hlavního programu

Tyto inicializace nezajistí otevření soketů. Navíc je nutné definovat proměnné pro každý použitý soket. Definici provádí až zde, protože dříve není k dispozici počet portů a počáteční TCP port načtený z externí paměti. TCP porty jsou přiděleny sekvenčně od počátečního portu, ten je vždy přidělen soketu pro vzdálenou správu. Sokety jsou umístěny do pole tak, že soket na pozici 0 odpovídá koncovému portu 1 atd. Mapování soketů na porty je vypsáno na lokální konzoli při zapnutí a také na řídicí soket po připojení k němu. Důležité je, že po změně počtu portů nebo počátečního čísla TCP portu je nutné provést restart zařízení, aby došlo ke korektní inicializaci.

```
TCP_SOCKET mgmtSocket;
TCP_SOCKET portsSockets[AppConfig.PortsCount];

mgmtSocket = MgmtTCPInit(AppConfig.MgmtTCPPort);
for(i = 0; i < AppConfig.PortsCount; i++) {
   portsSockets[i] = PortTCPInit(AppConfig.MgmtTCPPort + 1 + i);
}
```

Výpis 9: Otevření potřebných TCP soketů

Na řadu přichází hlavní smyčka, ta musí volat metody zpracovávající příchozí provoz. Výpis 10 zobrazuje stručný obsah hlavní smyčky. Na začátku jsou funkce obsluhující TCP/IP. Funkce *StackTask* dělá nízko-úrovňové klientské operace. Přesouvá data ze vstupního bufferu řadiče do příslušných front, zpracovává ARP dotazy, řídí DHCP klienta a jiné. Druhá obsluhuje serverové služby (HTTP, FTP, SMTP). Nepoužívám ani jednu z výrobcem dodaných služeb, tak funkce vlastně nic nedělá. Nechal jsem ji ovšem zde, kdyby se v budoucnu některá ze služeb použila. V každém cyklu se provede volání obsluhy aktivního portu, tam probíhá výměna dat s koncovým MCU. Poslední část smyčky slouží

k výpisu aktuální adresy. V případě, že dojde ke změně IP adresy, bude vypsána na konzoli. Hlavní cyklus obsahuje ještě některé funkce pro obsluhu konfiguračního módu a řídicího soketu.

```
StackTask();
StackApplications();

if (activePort == AppConfig.PortsCount) activePort = 0;
ProcessPort(++activePort, portsSockets[activePort − 1]);

if (dwLastIP != AppConfig.MyIPAddr.Val) {
    dwLastIP = AppConfig.MyIPAddr.Val;

    putrsUART((ROM char∗)"\r\nNew IP Address: ");
    DisplayIPValue(AppConfig.MyIPAddr);
    putrsUART((ROM char∗)"\r\n");
}
```

Výpis 10: Hlavní smyčka programu

Pro práci se sokety je k dispozici několik užitečných metod. Výpis 11 obsahuje prototypy těchto funkcí. Rozdělil jsem je do 3 skupin podle jejich funkce. První skupina má jen jednu funkci, která slouží pro detekci připojeného klienta. Pokud funkce vrátí *pravda*, znamená to, že na daný soket je připojen klient. Druhá skupina se používá k získání počtu dat. *TCPIsGetReady* vrací počet nezpracovaných bajtů ve vstupním bufferu. *TCPIsPutReady* funguje pro obrácený směr a vypisuje tedy počet volných bajtů ve výstupní vyrovnávací paměti. Je nutné manuálně kontrolovat, zda se zapisovaná data vejdou, aby nedošlo k přetečení. Poslední skupina obsahuje funkce pro přímou manipulaci s daty. Jedna slouží ke čtení dat a druhá k zápisu. Jako argument se udává ukazatel, kam se mají data zapsat, případně odkud se mají přečíst. Musíme uvést i jejich počet, který získáme z předešlé skupiny.

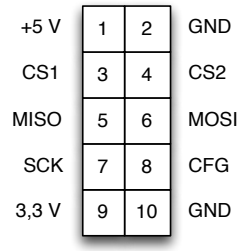
```
BOOL TCPIsConnected(TCP_SOCKET hTCP);

WORD TCPIsGetReady(TCP_SOCKET hTCP);
WORD TCPIsPutReady(TCP_SOCKET hTCP);

WORD TCPGetArray(TCP_SOCKET hTCP, BYTE∗ buffer, WORD len);
WORD TCPPutArray(TCP_SOCKET hTCP, BYTE∗ data, WORD len);
```

Výpis 11: Prototypy funkcí pro práci s TCP

Zařízení funguje pouze jako TCP server. Nenavazuje sám sokety na nějakou IP adresu. Místo toho pouze čeká, než se připojí klient.

5.2 Interní komunikace

V systému jsou 2 SPI sběrnice. V obou hraje roli Master PIC32 disponující dvěma moduly. SPI1 je určené pro komunikaci s obvody souvisejícími s TCP/IP. Takové obvody jsou celkem 2. Nejčastěji probíhá komunikace s řadičem Ethernet ENC28J60. Tento obvod umí komunikovat po SPI až na rychlosti 20 MHz. Druhý obvod je paměť EEPROM



Obrázek 3: Zapojení propojovacího konektoru

s maximální rychlostí 10 MHz. Obsluha tohoto modulu je čistě v režii TCP/IP Stacku. Ten pro zefektivnění komunikace mění svou rychlost podle zařízení, se kterým komunikuje. S pamětí probíhá komunikace jen při spuštění, kdy se načítá adresa a konfigurace, a při ukládání konfigurace.

Komunikace koncových modulů s řídicím MCU využívá také protokol SPI. Frekvence hodinového signálu je 5 MHz. U takto vysoké frekvence už záleží na délce vodiče mezi uzly, proto se snažím držet vedení co nejkratší. Při sledování průběhu na osciloskopu jsem sice pozoroval zákmity způsobené parazitními vlastnostmi vedení, to ale nevadilo úspěšné komunikaci. Jednotlivé desky koncových portů se propojují s hlavním MCU pomocí konektoru z obrázku 3. Konektor se nasazuje na plochý 10žilový kabel. U něj je právě důležitá délka. SPI piny jsou společné pro oba čipy umístěné na modulu. Každý koncový čip má svůj vlastní $\overline{SS}$ a je přímo připojen na sběrnici. MCU starající se o koncové porty mají také integrovaný SPI modul. Tento modul bohužel umí pouze 8bitovou komunikaci. V tomto konkrétním případě by bylo lepší použít 9bitovou s nejvyšším bitem určujícím data nebo příkazy. Chtěl jsem i přesto mít možnost měnit parametry koncových modulů z hlavního čipu. Problém jsem vyřešil použitím dalšího signálu, který jsem dovedl ke každému MCU. Signál má označení CFG a pokud je v logické 1, jsou veškerá data na lince řídicí. Vzdálená konfigurace není v současné verzi firmwaru implementována, avšak je hotová HW příprava této funkcionality. Vytvořit potřebný spoj bylo potřeba v době návrhu PCB. SW se dá jednoduše nahrát, ale navrhnout, vyrobit a osadit novou desku není zrovna nejjednodušší.

Po osazení všech PCB jsem narazil na problém s rušením, kdy řídicí MCU přijal data, která nikdo nevyslal. To v krajním případě způsobilo zacyklení programu při čekání na data, která nikdo neposlal. Bylo to v raných fázích vývoje, kdy nebyla ošetřena komunikace s porty pouze zapojenými. Následkem těchto objevů jsem přidal možnost nastavit počet portů. Komunikace tak probíhá jen s počtem portů, které se nastaví. Pro jistotu jsem ještě dodal podmínku, že hodnota prvního bajtu, který koncový modul pošle, musí mít hodnotu 21. Po nasazení výše uvedených opatření jsem již podobné problémy nezaznamenal.

Výpis 11 ukazuje princip komunikace s koncovým portem přes SPI, kód včetně obsluhy TCP/IP je uveden v příloze. Komunikace začíná nastavením příslušného $\overline{SS}$ do logické nuly. Tím modul v koncovém MCU začne naslouchat SCK a MOSI. Zároveň při každém cyklu hodin dojde k poslání jednoho bitu na MISO. Jako první informuje Master,

kolik bajtů má ve vstupním bufferu daného soketu. Tuto hodnotu zajistí funkce *TCPIsGetReady*. Při přenosu této hodnoty Slave pošle identifikační hodnotu 21. Pokud ji nepošle, nastaví se $\overline{SS}$ zpět do jedničky a obsluha se ukončí. V případě úspěšné detekce modulu proběhne načítaní dat ze Slave zařízení. K umožnění komunikace od Slave zařízení je nutné, aby Master generoval hodinový signál. To se provádí posláním neužitečných dat přes sběrnici. Na hodnotě takto poslaných dat nezáleží, protože je druhá strana stejně zahodí. Při prvním přenosu se získá počet bajtů, jenž chce poslat koncový MCU. Mezitím dojde na straně Slave kontrola, zda lze přijmout všechna data, která chce Master poslat. Na základě této kontroly Slave pošle počet kolik bajtů může přijmout. Jak bylo uvedeno dříve, je nutné posílat data z obou stran i v případě potřeby jen jednosměrné komunikace. Z tohoto důvodu je nutné určit celkový počet přenosů. Ten je roven většímu z čísel počtu k odeslání a přijetí. Mám také ochranu, aby přenos netrval déle, než dokáže koncový modul zaplnit svou vstupní vyrovnávací paměť. Následně začne samotný přenos dat. Musí se opět zohlednit užitečná a neužitečná data. Poslední krok je odpojení Slave nastavením $\overline{SS}$ zpět do jedničky.

```
SelectPort(activePort);

WriteSPI2(toSend);              // Count of byte to send
if (ReadSPI2() != 21) {         // Check for connected port
   DeselectPort(activePort);
   return;
}

WriteSPI2(0);                   // Write dummy value
toRecieve = ReadSPI2();         // Count of byte to receive

WriteSPI2(0);                   // Write dummy value
toSend = ReadSPI2();            // Free space in remote buffer

if (toSend > toRecieve)    totalPeriods = toSend;
else                       totalPeriods = toRecieve;
if (totalPeriods > 24)     totalPeriods = 24;

for(i = 0; i < totalPeriods; i++) {
   if (w < toSend)    WriteSPI2(sendBuffer[w++]);
   else               WriteSPI2(0);
   if (r < toRecieve) readBuffer[r++] = ReadSPI2();
   else               ReadSPI2();
}

DeselectPort(activePort);
```

Výpis 12: SPI komunikace na straně Master

Na straně koncového portu je situace diametrálně odlišná. Nemůže si totiž určit čas odeslání dat, protože ten určuje Master. Interní SPI modul umí vyvolat přerušení při přenesení bajtu dat. Není to ale zcela ideální, protože přerušení je vyvoláno vždy až po přenesení celého bajtu. Nelze tak detekovat úplný začátek přenosu. Jelikož potřebuji jako první bajt poslat hodnotu 21, aby Master detekoval existenci koncového portu, tak musím

hodnotu do vyrovnávací paměti SPI modulu zapsat již v hlavní smyčce před začátkem přenosu. Zbytek komunikace se řeší v obsluze přerušení, rozdělil jsem ji do 3 stavů. Průchod stavy je ukázán na výpisu 11. Po přijetí prvního bajtu se dočasně uloží přečtená hodnota a připraví se k odeslání počet načtených dat ve vstupním bufferu. Samotné odeslání proběhne paralelně s příjmem dalšího bajtu. Potom se zpracuje přečtený bajt od hlavního čipu obsahující počet dat, která chce Mater poslat. Ověří se, zda je pro ně ve výstupním bufferu dostatek místa. Následuje přechod do druhého stavu. Při přijetí dalšího bajtu se jeho vyčítání provádí jen pro zabránění nastavení příznaku přepsání vstupního bufferu. V této fázi se pouze připraví na odeslání dříve vypočítaná hodnota počtu dat k přijetí. Program přejde do mezifáze, kdy vyčte nulu poslanou Mastrem, protože potřeboval vyčíst hodnotu volného místa v bufferu, ale Master v dalším cyklu očekává data z vyrovnávací paměti. Musím je tedy připravit do bufferu SPI modulu. Poté už následuje pouze přenos užitečných dat. Je opět nutné oddělovat užitečná od neužitečných. Existuje možnost, kdy se délka přenosu nevolí podle největšího množství dat, ale zmenší se na hodnotu menšího počtu. V každém taktu jsou přenášena obousměrně užitečná data. V této aplikaci zmíněný přístup nejde použít. Například při zapnutí síťového prvku dojde k vytvoření velkého množství jednosměrných dat, která by se tak nemusela přenést. Nepoužívám zde funkce pro práci s SPI jako na straně PIC32, protože program musí být vykonán co nejrychleji a každý odskok způsobuje nemalé zdržení.

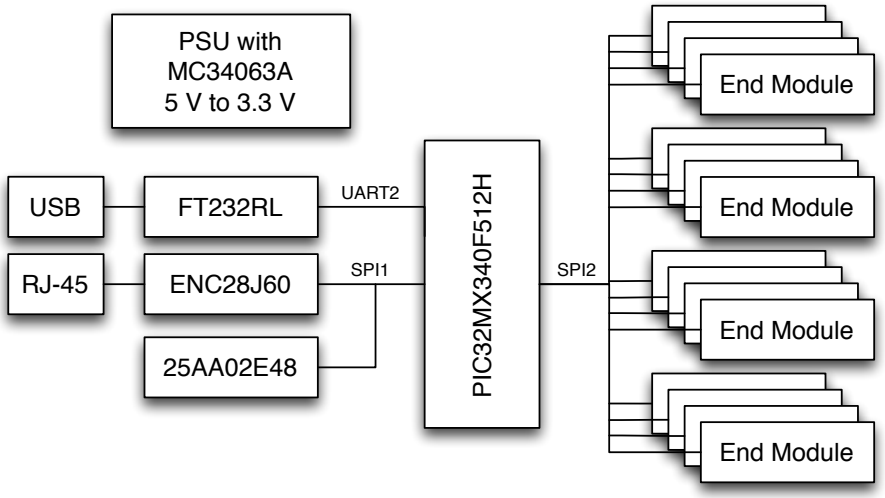
```
switch(status) {
    case SPI_START:
        rcvd = SSP1BUF;             // Read master buffer value
        counter_to_transmit = counter_r;
        SSP1BUF = counter_to_transmit;

        if (rcvd < 255 - counter_t)    counter_to_receive = rcvd;
        else                           counter_to_receive = 255 - counter_t;

        status = SPI_INIT;
        break;

    case SPI_INIT:
        rcvd = SSP1BUF;          // Read dummy value
        SSP1BUF = counter_to_receive;
        status = SPI_INIT2;
        break;

    case SPI_INIT2:
        rcvd = SSP1BUF;
        if (counter_to_transmit) {
            SSP1BUF = buffer_input[index_r++];
            counter_r--;
            counter_to_transmit--;
        }
        status = SPI_DATA;
        break;

    case SPI_DATA:
        if (counter_to_receive) {
```

```
            buffer_output[index_t++] = SPIRead();
            counter_t++;
            counter_to_receive--;
        }
        if(counter_to_transmit) {
            SPIWrite(buffer_input[index_r++]);
            counter_r--;
            counter_to_transmit--;
        }
        break;
    }
```

Výpis 13: SPI komunikace na straně Slave

Na straně Master jsem nemusel implementovat vlastní vyrovnávací paměť a práci s ní, mohu totiž rovnou využít FIFO TCP soketu, ale u koncového modulu již nic takového k dispozici není. Vytvořil jsem dva buffery pro dočasné uložení dat. Pro práci s každým bufferem používám 2 ukazatele a jedno počítadlo. Pro vstupní vyrovnávací paměť je jejich použití následující. Jeden ukazatel určuje místo v bufferu, kde se vloží hodnota z modulu UART. Druhý ukazuje na pozici s bajtem k odeslání přes SPI. V počítadle je pak počet dat, která jsou přečtená přes UART a čekají na odeslání přes SPI. I když má přenos dat z bufferu koncového modulu do hlavního MCU smysl jen při aktivním spojení na daném soketu, není možné nechávat data v koncovém modulu. Interní komunikace není ovlivněna stavem soketů. Jediný rozdíl v obsluze spočívá v absenci zápisu do výstupního FIFO bufferu soketu. Data jsou tedy zahazována na straně hlavního MCU až po jejich přijetí. Při této metodě nemusí dostávat koncový modul informaci o stavu soketů, aby mohl data zahazovat před zápisem do vyrovnávací paměti.

Při implementaci obsluhy SPI na straně koncového modulu jsem narazil na komplikaci s rychlostí zpracování příchozích dat. Použitý čip nemá architekturu optimalizovanou pro programování v jazyce C a navíc provozuji kompilátor v neregistrované verzi, jenž nemá možnosti optimalizace kódu. Tyto dvě vlastnosti se podepsaly na rychlosti, následkem čehož čip není schopen zpracovávat data dostatečně rychle. Dochází k přepisu dat před jejich vyčtením. Přitom frekvence sběrnice nedělá čipu problémy, ale obsluha přerušení trvá příliš dlouho. V původním návrhu jsem plánoval taktovat čip z energetických důvodů na 8 MHz. Po zjištění problémů s výkonností jsem v prvním kroku změnil hodnotu oscilátoru na 32 MHz, což sice pomohlo, ale nevyřešilo problém úplně. Opravdu funkční řešení bylo vložení zpoždění 10 μs za každý odeslaný bajt na straně PIC32. Po této změně již provoz funguje spolehlivě, ale přineslo to omezení ze strany rychlosti koncových portů. Nejde již tedy dosáhnout rychlosti 115 200 Baudů a 32 portů. Původně jsem počítal s přenosem přibližně 240 Bajtů za 0,5 ms. Samotný přenos při taktu 5 MHz trvá 1,6 μs. Z toho vyplývá, že jeden bajt se přenese každých 11,6 μs. Lze snadno spočítat, že za 0,5 ms dojde k přenesení 40 B. To samozřejmě vede k úpravě parametrů celého systému. Omezil jsem rychlost na maximálně 19200 Baudů. Je to sice omezení, ale stále je zařízení vhodné pro cílenou aplikaci ve správě aktivních síťových prvků.



Obrázek 4: Blokový diagram hlavního modulu

5.3 Hlavní modul

Deska hlavního modulu má několik částí propojených do jednoho celku. Nejlépe to vystihuje blokový diagram zobrazený na obrázku 4. I když většina komponent vyžaduje napájení 3,3 V, zvolil jsem za hlavní napětí 5 V. Nejpoužívanější výstupní budiče jsou dělané právě na toto napětí. Existují samozřejmě i verze pro 3,3 V, ty jsou ale nesrovnatelně dražší a hůře k sehnání. Vyšší hlavní napětí se daleko jednodušeji transformuje na nižší. Obvykle se převod provádí lineárním stabilizátorem. Je to sice ověřené řešení, ale dnes už bývá nahrazováno spínanými měniči. Hlavní rozdíl je v účinnosti převodu, a s tím souvisejícím tepelným vyzařováním. Veškeré ztráty se totiž projevují jako odpadní teplo. S takto vznikajícím teplem je nutno počítat v celém návrhu. Neprojevuje se totiž jen u součástky, která ho generuje, ale i ostatních částech systému. Když použijeme chladič, tak zlepšíme tepelné namáhání jedné součástky, ale oteplíme obsah celé krabice. Chladič slouží pouze k rozptýlení tepla přes co největší plochu do okolního vzduchu. Teplý vzduch se následně může hromadit a způsobit zahřátí celého zařízení. Proti této situaci se instaluje ventilátor, který teplý vzduch rozpohybuje. Po zhodnocení všech těchto negativních aspektů použití lineárního stabilizátoru jsem se rozhodl použít spínaný sestupný měnič. Použil jsem obvod MC34063A firmy ON Semiconductor, který patří k nejpopulárnějším řídícím obvodům pro takovéto aplikace. Je to opravdu univerzální součástka použitelná nejen pro sestupný, ale také vzestupný a invertující měnič. V katalogovém listu jsou zapojení pro jednotlivé převody. Jsou tam k nalezení i vzorce potřebné k výpočtu hodnot použitých součástek. Navrhovaný zdroj má vstupní napětí (V_{in}) 5 V a výstupní napětí (V_{out}) 3,3 V. Pro výpočet je nutné stanovit minimální vstupní napětí ($V_{in(min)}$), to se většinou rovná 90 % vstupního napětí. V mém případě má hodnotu 4,5 V. Saturační napětí (V_{sat}) interního spínacího tranzistoru je uvedeno v elektrických specifikacích řídicího obvodu měniče a má typickou hodnotu 1 V. Poslední nutný parametr je úbytek napětí

na diodě v propustném směru (V_F) při nominální hodnotě proudu (0,4 A), kterou lze najít pomocí Volt-Ampérové charakteristiky zvolené diody. Použil jsem diodu 1N5819 a z charakteristiky jsem vyčetl úbytek napětí 0,4 V. Měnič má maximální provozní frekvenci 100 kHz a s vyšší frekvencí je potřeba menší cívky, zvolil jsem proto 100 kHz. Protože jsem měl k dispozici kondenzátor 470 μF z dříve navrhovaného zdroje, použil jsem jej pro výstupní filtr zdroje. Hodnota tohoto kondenzátoru totiž určuje pouze výstupní zvlnění. Z toho vyplývá, že větší hodnota nemůže zdroji ublížit. Volba výstupního napětí se počítá z napěťového děliče. Jeden odpor v tomto děliči je nutné zvolit a druhý se potom dopočítává. Vybral jsem pro R_1 hodnotu 11 $k\Omega$. Následují výpočty obsahují všechny kroky nutné k určení hodnot součástek, které jsou součástí zdroje.

$$
\begin{aligned}
\frac{t_{on}}{t_{off}} &= \frac{V_{out}+V_F}{V_{in(min)}-V_{sat}-V_{out}} = \frac{3{,}3+0{,}4}{4{,}5-1-3{,}3} = 19{,}5 \\
(t_{on}+t_{off}) &= \frac{1}{f} = \frac{1}{1\cdot 10^5} = 1\cdot 10^{-5}s \\
t_{off} &= \frac{t_{on}+t_{off}}{\frac{t_{on}}{t_{off}}+1} = \frac{1\cdot 10^{-5}}{19{,}5+1} = 4{,}878\cdot 10^{-7}s \\
t_{on} &= (t_{on}+t_{off}) - t_{off} = 10^{-5} - 4{,}878\cdot 10^{-7} = 9{,}5122\cdot 10^{-5}s \\
C_T &= 4\cdot 10^{-5}\cdot t_{on} = 4\cdot 10^{-5}\cdot 9{,}5122\cdot 10^{-5} = 3{,}80488\cdot 10^{-10} \doteq \underline{380pF} \\
I_{pk(switch)} &= 2\cdot I_{out(max)} = 2\cdot 0{,}4 = 0{,}8A \\
R_{SC} &= \frac{0{,}3}{I_{pk(switch)}} = \frac{0{,}3}{0{,}8} = 0{,}375\Omega \doteq \underline{0{,}33\Omega} \\
L_{(min)} &= \left(\frac{V_{in(min)}-V_{sat}-V_{out}}{I_{pk(switch)}}\right)\cdot t_{on} = \left(\frac{4{,}5-1-3{,}3}{0{,}8}\right)\cdot 9{,}5122\cdot 10^{-5} = \\
&= 2{,}378\cdot 10^{-5} \doteq \underline{20\mu H} \\
C_0 &= \frac{I_{pk(switch)}\cdot(t_{on}+t_{off})}{8\cdot V_{ripple(pp)}} = \frac{0{,}8\cdot 10^{-5}}{8\cdot 0{,}002} = 5\cdot 10^{-5} \doteq \underline{470\mu F} \\
R_1 &= 11\cdot 10^3\Omega = \underline{11k\Omega} \\
R_2 &= R_1\cdot\left(\frac{V_{out}}{1{,}25}-1\right) = 11\cdot 10^3\cdot\left(\frac{3{,}3}{1{,}25}-1\right) = 18{,}04\cdot 10^3 \doteq \underline{18k\Omega}
\end{aligned}
$$

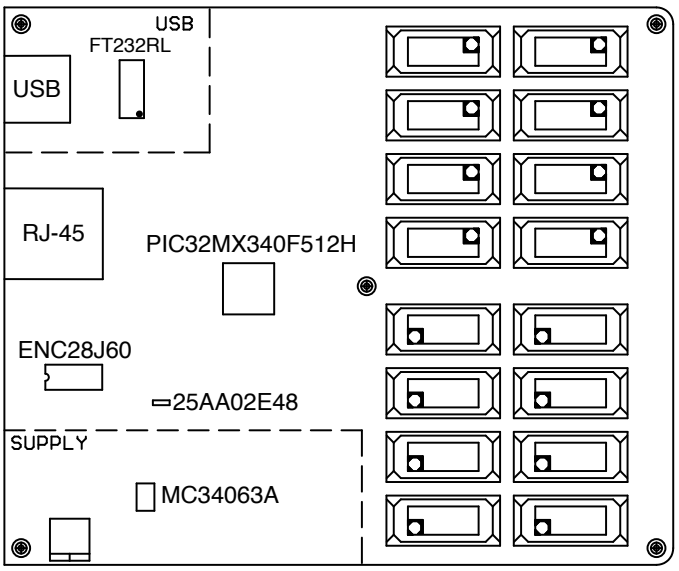
U spínaného zdroje je nebezpečí, že může docházet k šíření rušení přes společnou zem. Během návrhu je nutné tuto skutečnost zohlednit a zabránit takovému šíření. Nejjednodušší způsob je propojit zem zdroje se společnou zemí až v místě za výstupním filtrem zdroje. Z pohledu PCB jde tedy o 2 různé plochy mědi spojené cestou pouze v jednom místě. Předem je pak známá cesta proudu a je zajištěn průchod výstupním filtrem.

Hlavní MCU je PIC32MX340F512H [12, 13] taktovaný na 80 MHz. Jde o velmi výkonný 32bitový mikropočítač postavený na architektuře MIPS32 M4K. Čip má 64 vývodů, aby bylo k dispozici dostatek vstupně/výstupních bran pro řízení všech $\overline{SS}$ koncových portů. Použil jsem pouzdro TQFP64 určené k povrchové montáži. Tato práce byla moje první

aplikace TCP/IP na MCU, chtěl jsem proto mít k dispozici dostatek paměti. Tento konkrétní model nabízí 512 KB pro paměť programu a 32 KB pro paměť dat. Je zajímavé, že vycházel cenově lépe než čipy stejné řady s menší pamětí. Zapojení MCU nevyžaduje nic složitého, pouze stačí umístit ke každému páru napájecích pinů 100 nF keramický kondenzátor. Důležité je, aby napětí nejprve vedlo na kondenzátor a až poté na napájecí piny. Jádro je napájeno napětím 1,8 V. Čip ale obsahuje interní regulátor z 3,3 V, který se zapíná přivedením na pin *ENVREG* logické jedničky. Pro stabilizaci výstupu interního stabilizátoru je vyžadován 10 μF kondenzátor připojený na pinu *Vcap*. Poslední nutný pin k zapojení je $\overline{MCLR}$, který v případě přivedení logické nuly provede restart čipu. Tento pin se zapojuje přes odpor velikosti 10 kΩ na kladné napájecí napětí. Součástku je nutné před použitím naprogramovat. K této operaci jsem využil sériové programovací rozhraní známé pod zkratkou ICSP. Mohl jsem tak měnit program i po osazení na desku. Rozhraní požaduje celkem 5 pinů. Jde o synchronní sběrnici, takže jeden pin je určen pro hodinový signál a druhý pro data. Další dva piny jsou napájecí, slouží nejen k napájení obvodu během programování, ale i k detekci připojeného obvodu. Poslední signál má napětí přibližně 10 V a je připojený k *Vpp* sdílející fyzicky pin s $\overline{MCLR}$. Toto napětí je nutné pro zápis do flash paměti programu. Rozhraní potřebuje svůj vlastní konektor. Nepoužil jsem konvekční řešení pomocí kolíkové lišty. Namísto toho jsem na plošný spoj umístil pouze kovové kontaktní plošky. Programovací kabel je zakončen konektorem P20-0545R firmy Harwin, které disponuje pružnými kontakty. Při přiložení kabelu dojde k vodivému spojení. Toto řešení má ohromnou výhodu v prostorové úspornosti a umístění pouze v jedné vrstvě. Na druhou stranu to komplikuje ladění, kde je nutné dlouhodobější komunikace ladicího přípravku a MCU.

USB připojení je realizováno přes dříve zmíněný obvod FT232RL v pouzdru SSOP28, které nabízí kompromis mezi velikostí a pájitelností. Obvod nemá složité zapojení a v základu stačí pouze blokovací kondenzátory a jeden 100 nF kondenzátor pro stabilizaci interního 3,3 V regulátoru. Velmi tomu pomáhá integrace hlavního oscilátoru přímo do obvodu. Pro pokročilé funkce má čip několik pinů. Z továrny jsou předkonfigurovány 2 piny pro signalizaci komunikace. Jsou přímo určené k připojení dvou LED, kde každá indikuje jeden směr. Propojil jsem tento čip s interním modulem PIC32 za použití minimálního zapojení. Jsou zapojeny jen piny pro vysílání a přijímání, žádné jiné. Čip je napájen 5 V dodávanými přes USB. Tento obvod lze také konfigurovat přes USB pomocí nástroje dodaného výrobcem. Lze tak měnit obsah interní EEPROM paměti, kde jsou uloženy identifikační údaje obvodu, ale také například funkce přídavných pinů. Při připojení USB do PC dojde k úvodní komunikaci, kdy se vyměňuje i hodnota proudu, kterou zařízení bude požadovat po USB sběrnici. Tato hodnota je opět uložena v EEPROM a lze tak vytvořit elementární ochranu proti nadproudu. Z USB napájím pouze obvod FT232RL, který potřebuje okolo 15 mA, proto jsem nechal v paměti původní hodnotu 100 mA.

Důležitou součástí je řadič Ethernet ENC28J60, který je také v pouzdru SSOP28. Obvod podporuje Ethernet 10BASE-T. Pod touto zkratkou se skrývá Ethernet o rychlosti 10 Mb/s s použitím UTP kabelu jako transportního média. Kvůli správné a spolehlivé funkci musí být řadič galvanicky oddělen od zbytku vedení počítačové sítě. Pro splnění tohoto poža-

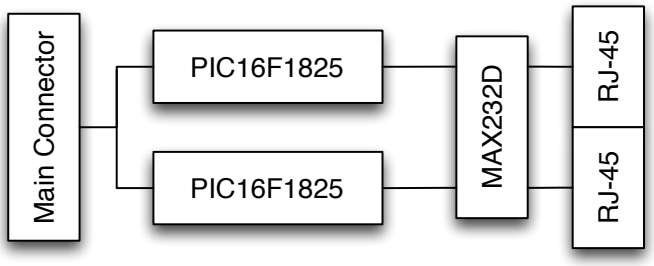


Obrázek 5: Rozmístění důležitých komponent hlavního modulu

davku se používá oddělovacích transformátorů. Pro ušetření místa na PCB jsem zvolil variantu konektoru RJ-45 s integrovaným transformátorem přímo v konektoru. Použil jsem konektor J0026D21BNL firmy Pulse Electronics. Ten má navíc integrované i LED. Řadič má vývody vyhrazené pro připojení dvou LED. Funkce zobrazení je programově nastavitelná a ve výchozím stavu jedna LED indikuje Link a druhá aktivitu. Zapojením LEDB se může určit výchozí duplex při připojení. Zvolil jsem zapojení pro poloviční duplex. Nastavení se dá změnit při konfiguraci Stacku. Opět jsem zapojení převzal z katalogového listu. Trochu nezvyklé je použití odporů s hodnotou 49,9 Ω a 2,32 kΩ, jenž jsou hůře k sehnání. Nečekaný problém byl sehnat oscilátor k tomuto řadiči s frekvencí 25 MHz.

Paměť 25AA02E48 má velmi úsporné pouzdro SOT-6. Protože má jen 6 vývodů, ze kterých 4 jsou určeny pro SPI, je jasné, že jediná vyžadovaná součástka bude blokovací kondenzátor mezi napájením. Tento integrovaný obvod lze napájet širokou škálou napětí. Výrobce uvádí 1,8 - 5,5 V. U ostatních součástek používám 3,3 V, stejné napětí tedy mohu použít i zde. Je to výhodné hlavně kvůli SPI, kde tak nemusím řešit převody mezi napěťovými hladinami. Paměť má předkonfigurovanou ochranu proti přepsání nejvyšší čtvrtiny své kapacity a to z důvodu zabránění přepisu EUI-48 adresy umístěné na konci tohoto bloku. Při sekvenčním zápisu, kdy se adresy automaticky inkrementují, stačí drobná nepozornost a dojde k přepsání celé paměti.

Pro připojení koncových modulů slouží celkem 16 konektorů MLW10G firmy Xinya určených pro montáž do PCB. Mají 10 pinů v konfiguraci dvou řad po 5. Navíc mají zámek znemožňující vložit kabelový konektor obráceně. Zásuvka k tomuto konektoru se nacvakává na plochý 10žilový kabel. Konektory jsem umístil do dvou řad. K usnadnění správy kabelů jsem půlku otočil o 180 stupňů, jak jde vidět na obrázku 5 znázorňujícím



Obrázek 6: Blokový diagram koncového modulu

celou desku. Zapojováním těchto konektorů vzniká velké fyzické namáhání plošného spoje, zejména způsobené prohnutím. Pro eliminaci takového chování jsem přidal v půlce řady konektorů další otvor pro šroub. Tato dodatečná podpora úplně zabránila prohýbání desky.

Celá deska má rozměr 100 x 120 mm. Nejde tedy zrovna o nejmenší PCB. Hlavní důvod k této velikosti je velké množství konektorů pro koncové moduly, které mezi sebou musí mít mezeru pro lepší manipulaci s kabely. Při návrhu jsem součástky rozdělil do logických celků, které jsem vyznačil přerušovanou čarou v servisním potisku. Každý blok je označen textem korespondujícím s jeho funkcí. K měření rušení zdroje jsem pro každou napěťovou hladinu použil testovací bod. Jeden takový bod je umístěn i na zem zdroje. Další testovací bod je na výstupu hodin z ENC28J60, protože jsem chtěl změřit, jaké jsou možnosti tohoto výstupu. Všechna testovací místa jsou popsána ve vrstvě servisního potisku v horní části desky. Není tak nutné hledat schéma, aby šlo zjistit, k čemu je daný bod určen. Signalizaci napětí provádím dvěma LED, které mají stejný předřadný odpor, aby šlo vidět rozdíl mezi napětím podle intenzity světla. Signalizace je také uvnitř hlavního zdroje a v síťovém přepínači na čelním panelu. V případě výpadku napájení lze rychle dohledat, kde se objevil problém. Při návrhu jsem zohlednil i dodatečnou cenu za druhý servisní potisk. Pro lehčí osazování a opravy chci vždy mít u každé součástky její půdorys a jméno. Pro splnění této podmínky a neplacení dodatečných poplatků jsem musel umístit veškeré součástky do vrchní vrstvy spoje.

Výsledné PCB má revizi B, protože při výrobě prvního prototypu pokaždé došlo ke spálení hlavního MCU. Chyba nastala při tvorbě pouzdra součástky. Omylem jsem přehodil čísla několika pinů mezi sebou. Shodou okolností v zasažených pinech byly 2 napájecí větve hlavního MCU. Po několika pokusech o úpravu hotové desky muselo být přistoupeno k návrhu a výrobě nového kusu. Revize A tedy nebyla funkční. Protože se ale jedná stále o stejné schématické zapojení, jde o verzi 1.0.

5.4 Koncový modul

Celý koncept je navržen tak, aby mohl být koncový modul co nejjednodušší a tím i velmi levný. Jeho hlavní činností je převod dat z koncového protokolu na SPI. Primárně je volen za koncový protokol UART, jak již bylo řečeno dříve. Všechny konektory musí být

dostupné na čelním panelu. Na síťových prvcích toho docílili použitím konektoru RJ-45. Z tohoto řešení jsem se inspiroval i v případě mé konstrukce. Síťové prvky mají ještě jednu konstrukční výhodu. Konektory Ethernet jsou vždy 2 nad sebou, aby bylo možné dosáhnout větší hustoty na 1 RU. Při počátcích vývoje agregátoru jsem měl v plánu pro každý port udělat vlastní PCB. Při použití jednoho plošného spoje by se totiž ztratila veškerá modularita. Problém ovšem byl, takovou desku uchytit a připojit k hlavní. Napadlo mě použít desky koncového modulu otočené o 90 stupňů. Dosáhl jsem tak relativně velké hustoty portů, protože mohly být od sebe umístěny na vzdálenost výšky konektorů. Nevýhoda takové úpravy byla nutnost použít desku širokou 4 cm, aby ji bylo možno dobře uchytit. Na takovém PCB bylo velké množství nevyužité plochy. Při pokusech umístit vedle sebe 2 samostatné konektory jsem měl strach ze zlomení čelního panelu z důvodu příliš úzkých zbytků plastu v místě konektoru. Našel jsem několik dvojitých konektorů, ale byly dražší (více než 50 Kč) než zbytek PCB. Náhodou v době návrhu zařadila firma GM Electronics s.r.o. do svého sortimentu duální RJ-45 konektor s označením WEBP 8-8 SHIELDED DVOJITY. Ten stál přibližně 15 Kč, což je velmi dobrá cena. Pro snazší identifikaci portů jsem ke každému konektoru umístil do servisního potisku číslici s pořadím v rámci PCB. Po vzoru aktivních prvků zachovávám konvenci, že port 1 je umístěn v levém horním rohu.

Další úkol byl vybrat koncové MCU. Chtěl jsem, aby se jednalo o PIC, protože mi přišlo praktické vyvíjet všechny firmwary z jednoho IDE. Čip toho nemusel umět hodně, stačilo mít HW modul pro SPI a UART. Protože čip nepotřeboval časovat SPI, tak mi bohatě stačil i integrovaný oscilátor. Toto rozhodnutí opět vedlo k finančním úsporám a navíc se dost zjednodušil návrh PCB. Důležitý aspekt při výběru byla cena. Zařízení má obsahovat 32 těchto MCU, takže se do výsledné ceny výrazně promítne každá koruna navíc za tyto MCU. Nutno bylo ještě zohlednit paměťové nároky vyrovnávacích pamětí. Původní výpočty počítaly s 32 porty s rychlostí až 115 200 Baudů. Komunikaci s každým koncovým portem jsem vyčlenil maximálně 0,5 ms. Z toho lze snadno určit, že do vyrovnací paměti se musí vejít 16 ms při maximální rychlosti komunikace. Ve výsledku to dá hodnotu 231 B. To je však hodnota jen pro jeden směr. Takže jsem hledal čip s 512 B paměti dat. Našel jsem sérii 8bitových MCU, které byly velmi levné (kolem 30 Kč) a měly pouzdro SOIC14. Takové pouzdro je běžné u logických obvodů, avšak u MCU jej vidím prvně. Bylo na výběr 256 B nebo 1024 B paměti. Samozřejmě jsem zvolil verzi s vetší pamětí PIC16F1825 [14]. Díky tomuto rozhodnutí jsem měl velkou paměťovou rezervu a nebylo nutné řešit každý použitý bajt. Důsledek se projevil i při tvorbě vyrovnávací paměti. Použitím datového typu *unsigned char* jsem si ulehčil práci s ukazateli nad těmito buffery. Při inkrementaci čísla 255 dojde k jeho automatickému vynulování. Jediná podmínka, aby vše správně fungovalo, je velikost celého bufferu 256 B. To mě moc netížilo při velikosti volné paměti. Nemusel jsem se bát ani přetečení v důsledku přijetí bajtů z UART během inicializace komunikace přes SPI.

Na straně UART komunikace bylo nutné provést převod na úrovně podle normy RS-232C. Tato činnost je již dlouhou dobu doménou integrovaných obvodů MAX232 obsahujících 2 nábojové pumpy. Zajistí tak zdvojení napájecího napětí (5 V) a jeho invertování. Pro logickou nulu bude výstupní napětí +10 V a pro logickou jedničku -10 V.

Tyto obvody byly původně vyvinuty firmou Maxim Integrated. Po několika letech jej ale začalo vyrábět více výrobců. Já jsem použil obvod s označením MAX232D firmy Texas Instruments. Nechci používat řízení toku, protože zařízení připojené do koncových portů ho nepodporují, tak jsem si mohl dovolit použít pouze jediný obvod MAX232. Tyto obvody mají totiž integrovány celkem 4 brány. 2 slouží pro převod úrovní při příjmu, druhé dvě při odesílání. Zapojil jsem tedy oba obvody PIC na jeden budič a ušetřil místo na PCB a také snížil cenu celého modulu. Důležité je i zapojení samotného konektoru. Navrhl jsem zapojení umožňující připojit konzoli aktivního síťového prvku za pomoci běžného přímého UTP kabelu.

PCB neobsahuje žádné otvory pro montáž, počítá se s umístěním do slotu. Výšku jsem určil 40 mm. Vím, že se běžně do 1 RU krabice umisťuje 40mm ventilátor, takže tato výška nemůže být na škodu. S délkou to ale nebylo tak jednoduché. V rámci této práce budou koncové moduly umět jenom UART. Tento protokol nevyžaduje nikterak složitou implementaci fyzické vrstvy komunikace. Bez problému by šlo vytvořit desku o délce jen 40 mm. Na druhou stranu jsem musel zohlednit možný budoucí vývoj těchto modulů. Montáž počítá s jednotnou velikostí všech PCB v zařízení. Kdybych se snažil udělat desku co nejmenší, tak může nastat problém při vytvoření modulů pro složitější protokoly. Změřil jsem dostupné místo v krabičce a nakonec určil délku na 60 mm. Firmware PIC32 je bez problémů schopen pracovat s koncovým modulem nesoucím pouze jeden port. Při nepřijetí identifikace se prostě chybějící port přeskočí.

Tato deska byla navrhnuta několik měsíců před hlavním PCB. Navíc šlo o mou prvně komerčně vyráběnou desku, proto si nese jisté chyby způsobené nezkušeností s touto výrobou. Hlavní nevýhoda je umístění součástek do spodní vrstvy a z toho vyplývající nutnost oboustranného servisního potisku. Navíc u programovacího konektoru nastal problém s klíčovacím otvorem, který byl prokoven a měl menší výsledný průměr než v návrhu. Musel jsem explicitně otvory na všech PCB zvětšit. U koncových modulů byl problém s programováním. I když jsem do všech modulů nahrával odzkoušené verze FW. Stalo se mi, že jsem je musel přehrát na novější verzi. V případě 4 koncových modulů to znamenalo 8x programovat čip pro každou verzi. Tato činnost, leč se nezdá, je ve výsledku velmi zdlouhavá a rapidně zpomaluje vývoj. Bohužel obvody PIC zatím nemají možnost aktualizace FW přes rozhraní I2C nebo SPI.

6 Klientská aplikace

Od začátku jsem se chtěl vyvarovat tvorbě komplikovaného protokolu pro komunikaci s agregátorem. Cílem byla co největší transparentnost přenosu. Výsledek je, že se v soketu přenášejí pouze surová data z jednoho konce na druhý. S agregátorem může komunikovat jakákoliv aplikace přes obyčejný TCP soket. V případě konzolového serveru projektu Virtlab, jenž má ve výsledku se zařízením komunikovat, půjde jen o předávaní dat z jednoho soketu do druhého. Před napsáním vlastní aplikace jsem používal aplikaci telnet. Tato aplikace umožňuje upravit své výchozí chování. Po spuštění odesílá text až po ukončení řádku klávesou *enter*. Velmi nepříjemné je v mém případě zobrazování rozepsaného textu na obrazovku přímo aplikací telnet. Další nevýhodou je posílání konce řádků jako <CR><NUL>.

Aplikace má konfigurační řádek, do kterého se dá přepnout stiskem kláves CTRL a]. V tomto režimu se píšou níže uvedené příkazy. První problém řeší přepnutí do znakového módu, takže se posílá každý znak ihned po stisknutí klávesy. To se provede příkazem *mode character*. Zakázání lokálního výpisu se dá dosáhnout zadáním *set echo off*. U posledního problému jsem nenašel možnost jej zcela eliminovat. Existuje sice možnost přes *set crlf* posílat ukončení řádku jako <CR><LF>. Ovšem aktivní síťové prvky očekávají pouze <CR>. Výsledkem bylo vkládání jednoho řádku navíc. V případě nouze jde ovšem toto řešení použít.

Napsal jsem si tedy vlastní aplikaci zobrazenou na výpise 14. Jako jazyk pro implementaci ukázky obsluhy jsem si zvolil Perl [15]. Je to můj oblíbený skriptovací jazyk, ve kterém jsem již práci se sokety vytvářel. Skript má 2 vstupní parametry. Prvním je adresa agregátoru, kde se chce klient připojit, druhý atribut určuje cílový TCP port. Aplikace v principu jen čte standardní vstup a posílá jeho obsah do TCP soketu a čte data ze soketu a vypisuje je na standardní výstup. Běžně při čtení vstupu od uživatele provede funkce provádějící čtení zablokování programu. Jinými slovy, pokud uživatel nezadá text, tak program stojí. Program čeká tak dlouho, než uživatel stiskne *enter*. Můj kód používá neblokované čtení ze standardního vstupu pomocí knihovny Term::ReadKey. Knihovna umožňuje také zamezit lokálnímu výpisu uživatelem zadaných znaků. Pro mou aplikaci je tato možnost velmi výhodná. Program bude posílat zprávu znak po znaku, přesně jako byl přímo připojen ke konzolovému portu pomocí sériového kabelu.

Čtení ze soketů trpí v Perlu stejným problémem s blokací jako načítání standardního vstupu. K řešení jsem ani nemohl použít dříve zmíněnou knihovnu, protože neumí pracovat se sokety. Vlastně mi stačí získat informaci, zda bylo něco načteno či nikoliv. K tomuto účelu se dá použít knihovna IO::Select. Nabízející funkci *can_read()*. Funkce má jako svůj atribut hodnotu v sekundách, jak dlouho má čekat než vrátí výsledek.

Aplikace je opravdu jen ukázkou, jak by se dala obsluha implementovat. Má i své chyby. Neumožňuje přenášet speciální znaky zadané pomocí zkratek CTRL+C a CTRL+Z. Ty mají v operačním systému většinou nadřazený význam. První z nich se dá použít k ukončení této aplikaci a odpojení od agregátoru.

Konzole aktivních prvků očekávají na vstupu jistou konvenci. Především jde o ukončování řádků jen za pomoci <CR>. Může nastat situace, kdy uživatelský terminál posílá pro ukončení jinou sadu znaků, proto jsem pomocí nástroje *stty* na začátku skriptu nastavil,

aby se posílalo jenom <CR>. V kódu volám program *stty* celkem dvakrát. Druhé volání souvisí s tím, že terminály často posílají při stisku klávesy **backspace** znak <DEL>. Tomu ale opět konzole aktivního prvku nerozumí. Druhé volání nastaví, že při stisku klávesy *backspace* se korektně pošle znak <BS>. Výsledkem je uživatelský komfort, jako kdyby šlo o připojení do fyzického COM portu počítače.

```
system('stty erase ^ && stty inlcr');

my $aggr_ip = $ARGV[0];
my $aggr_port = $ARGV[1];

my $socket = IO::Socket::INET->new(    PeerAddr => $aggr_ip,
                                       PeerPort => $aggr_port,
                                       Proto    => "tcp",
                                       Type     => SOCK_STREAM,
                                       Timeout  => 1)
   or die "Couldn't connect to $aggr_ip:$aggr_port : $@\n";

my $select = IO::Select->new();
$select->add($socket);

my $readed = "";
my $recieved;

while(1) {
   ReadMode('cbreak');
   if (defined ($readed = ReadKey(-1))) {
      print $socket $readed;
   }
   ReadMode('noecho');
   if ($select->can_read(.01)) {
      $socket->recv($recieved, MSG_DONTWAIT);
      print $recieved;
   }
}
close($socket);
exit;
```

Výpis 14: Klientská aplikace pro PC

7 Testování

Testování zařízení probíhalo v několika fázích už během vývoje. Při dokončení každého většího celku jsem musel otestovat, zda se neovlivnila funkčnost předchozích částí. Největší problémy se objevovaly po předání komunikace po SPI. Měl jsem ověřenou část obsluhující TCP/IP, ale stávalo se, že obsluha SPI způsobila zacyklení programu a celé TCP/IP se zaseklo.

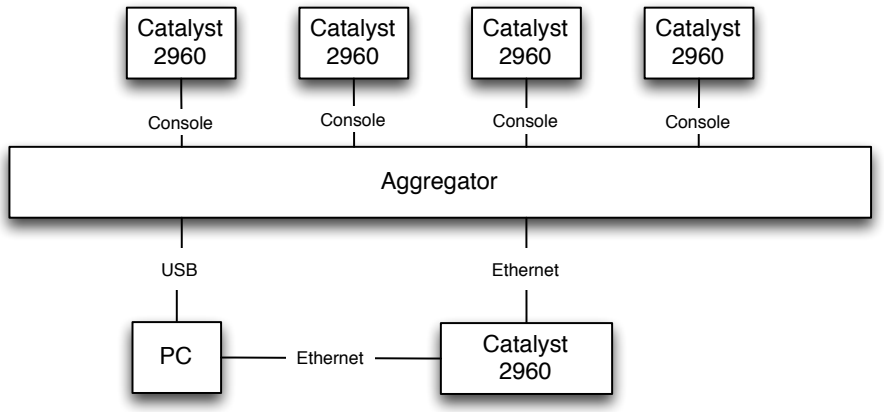
Hlavní úkol testování byl ověřit chování zařízení v cílovém prostředí. Vývoj probíhal mimo školu, ale výsledné testování muselo proběhnout ve škole. Rozdělil jsem testování na 2 fáze.

Během první šlo především o test funkčnosti interní komunikace, její rychlosti a spolehlivosti. Při tomto testu byl na straně klienta použit program telnet, který se ukázal velmi nevhodný pro tento typ aplikace. Kritická nevýhoda spočívala ve špatném posílaní zvláštních sekvencí, čehož následkem nešlo například probudit uspanou konzoli prvku. Testovací topologie je na obrázku 7. Zapojil jsem konzole 4 přepínačů Catalyst 2960, používající rychlost 9600 Baudů, do portů 1 až 4 na agregátoru. K agregátoru jsem se připojil přes USB pro správu. Skrz další přepínač jsem propojil mé zařízení s PC.

Navázal jsem 4 spojení přes telnet na nastavené TCP porty odpovídající použitým koncovým portům. Protože nefungovalo probouzení konzole přes tlačítko *Return* musel jsem každou konzoli probudit přepojením na COM port PC. Poté již vše fungovalo. K simulaci paralelního přístupu mi posloužila aplikace iTerm umožňující vkládat jeden vstup zároveň do několika terminálových oken. Mohl jsem v jeden okamžik pracovat se všemi konzolemi najednou. Toto řešení fungovalo bezvadně. Poslední věc k otestování bylo zahlcení ze strany konzole síťového prvku. K tomuto účelu jsem použil nejobsáhlejší výpis na těchto prvcích a to *show tech*. Ten se používá při řešení technických problémů přímo s výrobcem a obsahuje veškeré informace o zařízení, včetně aktivní konfigurace a obsahu pamětí. Samotný výpis trval kolem 5 minut. Všechny 4 konzole zobrazovali korektní obsah výpisu. Reagovaly jak během výpisu, tak i po jeho skončení. Tímto testem jsem ověřil, že nedochází ke ztrátě dat a zacyklení na straně koncového modulu. Tato fáze byla mířena hlavně na program koncových MCU. Zjistil jsem, že aplikace telnet je nedostačující pro mé použití.

Test probíhal v laboratoři počítačových sítí. Výhoda tohoto prostředí spočívá ve velkém množství aktivních síťových prvků firmy Cisco Systems, pro které má být zařízení primárně určeno. Pro ochranu konektorů konzolí jsou všechny vyvedeny do propojovacího panelu vedle sebe. To ulehčilo zapojení agregátoru, protože všechny kabely vedou z jednoho panelu.

Při testu se ukázalo několik chyb vzniklých během vývoje. Předpokládal jsem automatickou detekci duplexu řadiče Ethernet po připojení do sítě. Tato funkce ovšem není implementována. To jsem zjistil až po připojení do aktivního prvku, kde šlo duplex zobrazit. Další problém je s funkcí Auto IP. Ta v případě nedostupného DHCP serveru má nastavit výchozí adresu z rozsahu 169.254.0.0/16 určeného pro lokální komunikaci. Nepředpokládal jsem, že při vypnutém DHCP klientu a uložené statické adrese v EEPROM dojde k načtení automatické IP. Při vývoji jsem vždy použil konfiguraci přes DHCP. Po úpravě konfigurace a dalším testu jsem tyto problémy již nepozoroval.



Obrázek 7: Topologie prvního testu

Fáze 2 probíhala ve stejné místnosti, se stejnými prvky, kterých tentokrát bylo 8. V průběhu testu bylo použito plně duplexního režimu Ethernetu. Při tomto testu šlo především o test klientské aplikace a firmware hlavního modulu. Při prvním připojení skrze mou aplikaci jsem zjistil, že prvek reaguje stejně špatně jako v případě znakového módu telnetu. Protože to bylo velmi podezřelé, tak jsem začal hledat co to může způsobit. Připojil jsem ICD k MCU na koncovém modulu a dal breakpoint na řádek z výpisem na UART. Zjistil jsem, že v případě přijetí jen jednoho znaku nedochází ke korektnímu uložení znaku do vyrovnávací paměti k odeslání na UART. Komunikace mezi Masterem a Slavem je totiž posunuta o jeden bajt. Přehlédl jsem při tvorbě kódu, že Master pošle dvakrát neužitečnou nulu, ale v obsluze Slave jsem ji zohlednil jen jednou. Po doplnění této drobnosti začala komunikace skrze mou aplikaci fungovat. Nedařilo se mi však probudit uspanou konzoli prvku. Běžně stačí stisk klávesy *enter*. Zkusil jsem přístup skrze aplikaci telnet. Tam mi ovšem probouzení, narozdíl od prvního testu, fungovalo. Věděl jsem podle odchyceného provozu v programu Wireshark, že má aplikace při stisku klávesy *enter* posílá znak <LF> a nyní fungovalo probouzení, při takovém nastavení telnetu, kdy posílá <CR><LF>. Z toho jsem usoudil nutnost poslat pro probuzení <CR>. Moje aplikace přebírá nastavení znaků pro ukončení řádku z aktuálního nastavení terminálu. Toto nastavení se dá upravit pomocí nástroje *stty*. Po chvilce hledání jsem našel správný parametr a probouzení a odřádkování fungovalo bez problémů. Narazil jsem na problém s nefunkční klávesou *backspace*. V provozu odchyceném při vývoji klientské aplikace jsem viděl, že můj terminál odesílá při stisku této klávesy <DEL>. S podobným problémem jsem se již setkal dříve. Konzole prostě znak <DEL> ignoruje a vyžaduje pouze <BS>. Opět jsem našel parametry pro *stty*, jenž změní tento znak. Ve výsledku to znamenalo dosažení plné funkcionality této práce. Nyní jde zařízení bezproblému naplno využívat k zamýšlenému účelu.

8 Závěr

Jsem velmi rád, že jsem si zvolil tuto práci. Od původního návrhu do konečného produktu byla velmi dlouhá a často trnitá cesta, ale stálo to za to. Vyzkoušel jsem si mnoho nových technologií, se kterými jsem se chtěl již dlouho seznámit. Jako ve většině projektů není výstup zcela identický s původním návrhem. Na začátku jsem nevěděl, jaké komplikace mohou nastat, když není přenosové vedení správně nevrhnuté a jak se v celé záležitosti projeví fyzikální limity. Při zjištění problému s rychlostí koncových modulů jsem měl dlouho pocit, že práci nebude možné dokončit. Naštěstí jsem to nevzdal a našel cestu, jak vše vyřešit jen za cenu ústupků na rychlosti koncových portů.

Největší práci mi dala samotná fyzikální realizace. Přeci jen u vývoje SW už je jasné, jaké prostředky jsou k dispozici a s čím se dá pracovat. Na druhou stranu při vývoji HW je nutné počítat se vším, co by mohlo být potřeba v budoucnu. Změna programu je otázka chvilky, ale úprava HW může zabrat až týdny a nemalé finanční prostředky. Proto jsem rád, že při velké složitosti PCB došlo jen k jedné hardwarové změně. Velikost změny je irelevantní, vždy se musí udělat nová deska. Úprava se dá ještě udělat pomocí drátků a přerušení původních tras. Mám s touto metodou špatné zkušenosti a většinou je to jenom zdroj dalších problémů.

Velké překvapení byla komplexita implementace TCP/IP Stacku firmou Microchip Technology. Viděl jsem již dříve, že se tato platforma často používá, ale možnosti mě překvapily. Nebylo ani nutné strávit mnoho týdnů v dokumentaci popisující použiti jednotlivých knihoven. Všechny zdrojové kódy byly velmi dobře okomentované a přehledné. Sice jsem pro tuto práci použil jen zlomek možností této aplikační sady, ale při úvodní analýze jsem strávil dost času zkoušením různých ukázkových kódů. Vydavatel Stacku vyrobil přímo vývojovou desku pro naučení práce s TCP/IP. Tato deska nese označení PIC32 Ethernet Start Kit a hostovala první půlrok vývoje. Jediný rozdíl byl v použitém řadiči Ethernet. Hotový kit používá interní Ethernet řadič v čipu PIC32MX795F512H. Tento řadič nemá ale implementaci fyzické vrstvy a musí být použit externí obvod. Já jsem zvolil celý externí řadič. Mé řešení je sice pomalejší, ale velmi levné a jednoduché na implementaci. Řadič ENC28J60 lze navíc kombinovat s více řídicími MCU různých výrobců. Jde o jeden z nejlevnějších obvodů svého typu na trhu.

Navíc jsem si zde mohl vyzkoušet reálné fungování algoritmu plovoucího okna u TCP komunikace. U PC se dost složitě simuluje přeplnění vstupních bufferů soketů. Zde stačilo jen lehce zpomalit zpracovávání a šlo vidět, jak zařízení přijme data, která zaplní celý buffer, a posléze zařízení postupně posílá aktualizaci velikosti okna. I pro mne jako člověka se specializací na počítačové sítě bylo velmi poučné pracovat na takto nízké vrstvě. Lidé z oblasti IT si zvykají na obrovské množství paměti a vysoký výpočetní výkon. To vede k menšímu zájmu o optimalizace aplikací. Doporučuji každému programátorovi vyzkoušet si situaci, kdy má jen 1 KB paměti programu a RISC jádro taktované na 32 MHz. Je to velmi zajímavá zkušenost, jež je silně motivující psát efektivní kód.

Jedním z klíčových aspektů vývoje je cena. Snažil jsem se vždy volit vhodný kompromis mezi cenou, kvalitou a výkonností. Ne vždy se šetření vyplatilo, hlavně co se týče řídicích obvodů koncových modulů. Použít dražší a výkonnější obvody, problém bych neměl. Na druhou stranu by se ztratil smysl této práce. Ve výsledku se celé zařízení pro 8

portů dá vyrobit za přibližně 2 500 Kč. Každý další koncový modul se dvěma porty vyjde okolo 300 Kč.

Jde ve výsledku o dostupné zařízení, které není problém vyrobit. Dá se navíc konfigurovat počet portů. Začne se na malém počtu a když bude potřeba připojit další zařízení, jen se zapojí další koncové moduly. Pořád půjde o stejnou platformu. Velká výhoda je taky v univerzálnosti celého řešení. Přeci jen lze připojit koncový modul s podporou jakéhokoliv protokolu. Jen nesmí jít o nic vysokorychlostního, aby stíhala interní sběrnice přenášet data z vyrovnávací paměti před jejich přepsáním. Interní komunikační protokol je tak jednoduchý, že jej pochopí i začínající návrhář v této oblasti.

S výsledným zařízením jsem spokojen, jak po fyzické tak i po programové části. Myslím, že se jedná o dobrý důkaz toho, že složitou činnost může vykonávat v principu jednoduché zařízení. Při doladění návrhu vedení pro interní SPI sběrnici tak, aby nedocházelo k velkému zkreslení a vytvoření upravené krabice speciálně pro tento produkt, si reálně dokážu představit komerční výrobu tohoto zařízení.

Petr Havlíček

9 Reference

[1] BLANK, Andrew G. *TCP/IP Foundations.* San Francisco, Calif.: Sybex, 2004, xv, 284 p. ISBN 07-821-4370-9.

[2] ISO/IEC 7498-1. *Basic Reference Model: The Basic Model.* 1996.

[3] PLUMMER, C. David. *An Ethernet Address Resolution Protocol, RFC 826.* 1982. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc826.txt

[4] SATRAPA, Pavel *IPv6: internetový protokol verze 6.* 3., aktualiz. a dopl. vyd. Praha: CZ.NIC, 2011, 407 s. CZ.NIC. ISBN 978-80-904248-4-5.

[5] POSTEL, J. *Internet Control Message Protocol, RFC 792.* 1981. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc792.txt

[6] REYNOLDS, J. POSTEL, J. *Assigned Numbers, RFC 1700.* 1994. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc1700.txt

[7] POSTEL, J. *User Datagram Protocol, RFC 768.* 1980. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc768.txt

[8] DARPA *Transmission Control Protocol, RFC 793.* 1981. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc793.txt

[9] DROMS, R. *Dynamic Host Configuration Protocol, RFC 2131.* 1997. [cit. 2013-04-11] Dostupné z: http://www.ietf.org/rfc/rfc2131.txt

[10] Motorola, Inc. *SPI Block Guide.* 2003.

[11] PLÍVA, Zdeněk. *EAGLE prakticky: řešení problémů při běžné práci.* 2. vyd. Praha: BEN - technická literatura, 2010, 184 s. ISBN 978-80-7300-252-7.

[12] Microchip Technology Inc. *PIC32MX3XX/4XX Data Sheet.* 2011. ISBN 978-1613411490

[13] DI JASIO, Lucio. *Programming 32-bit Microcontrollers in C: Exploring the PIC32.* Burlington, MA: Newnes, 2008, xxiii, 527 p. ISBN 978-075-0687-096.

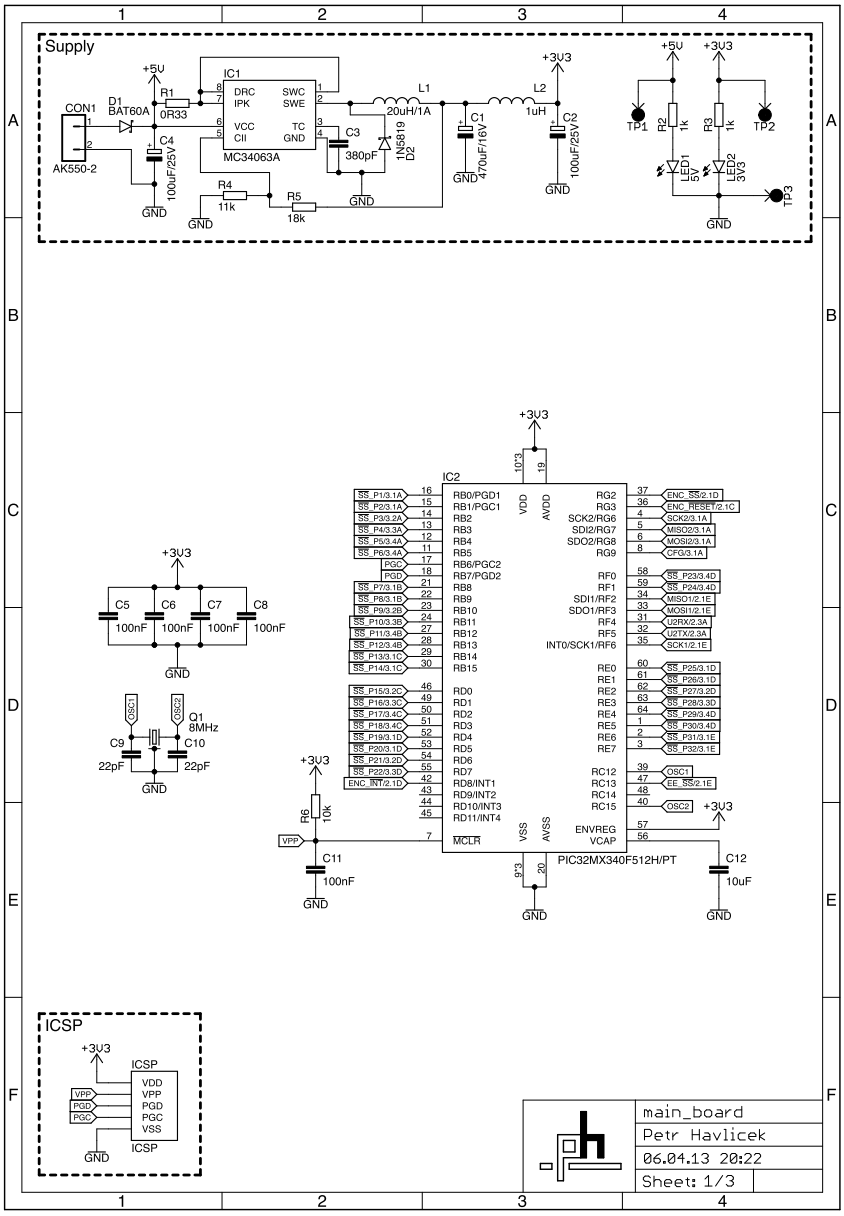
[14] Microchip Technology Inc. *PIC16(L)F1825/1829 Data Sheet.* 2012. ISBN 978-1620764039.

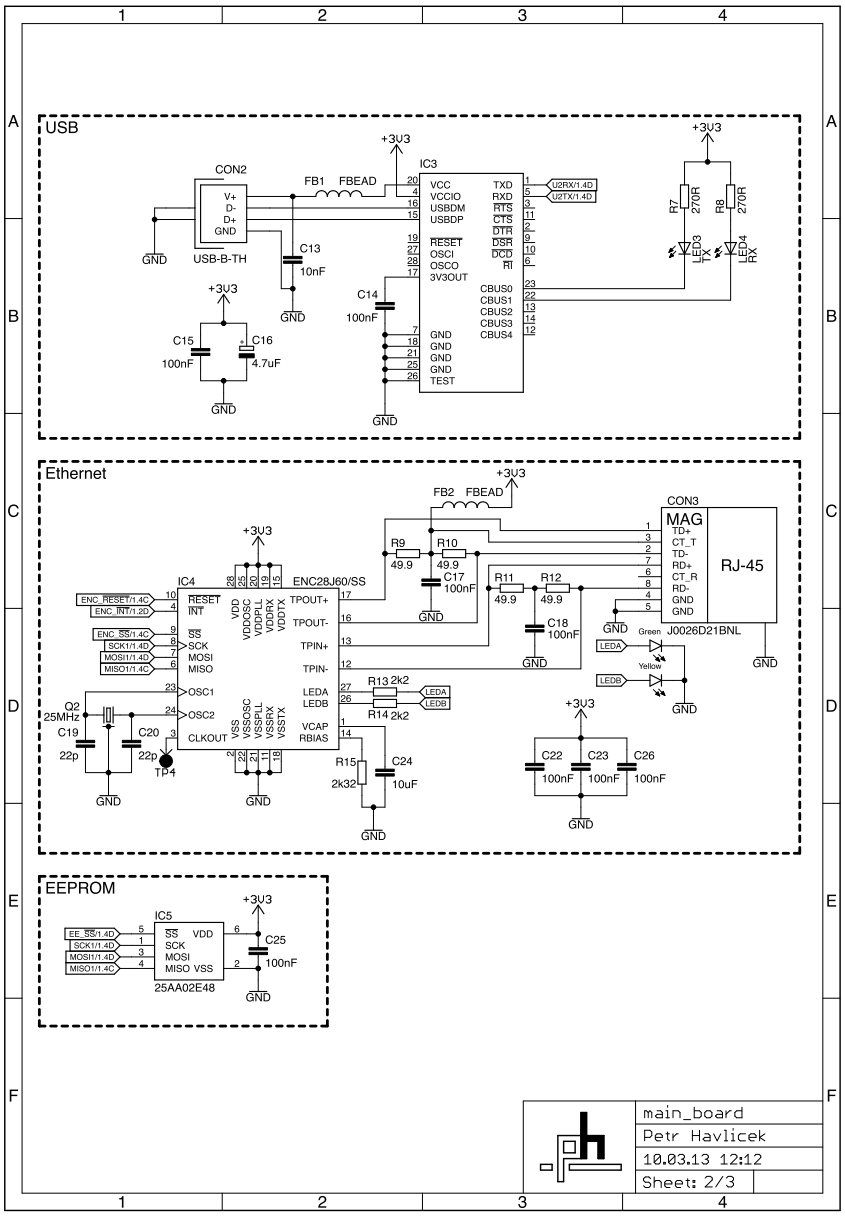
[15] WAINWRIGHT, Peter. *Pro Perl.* Berkeley, CA: Apress, 2005. ISBN 14-302-0014-6.

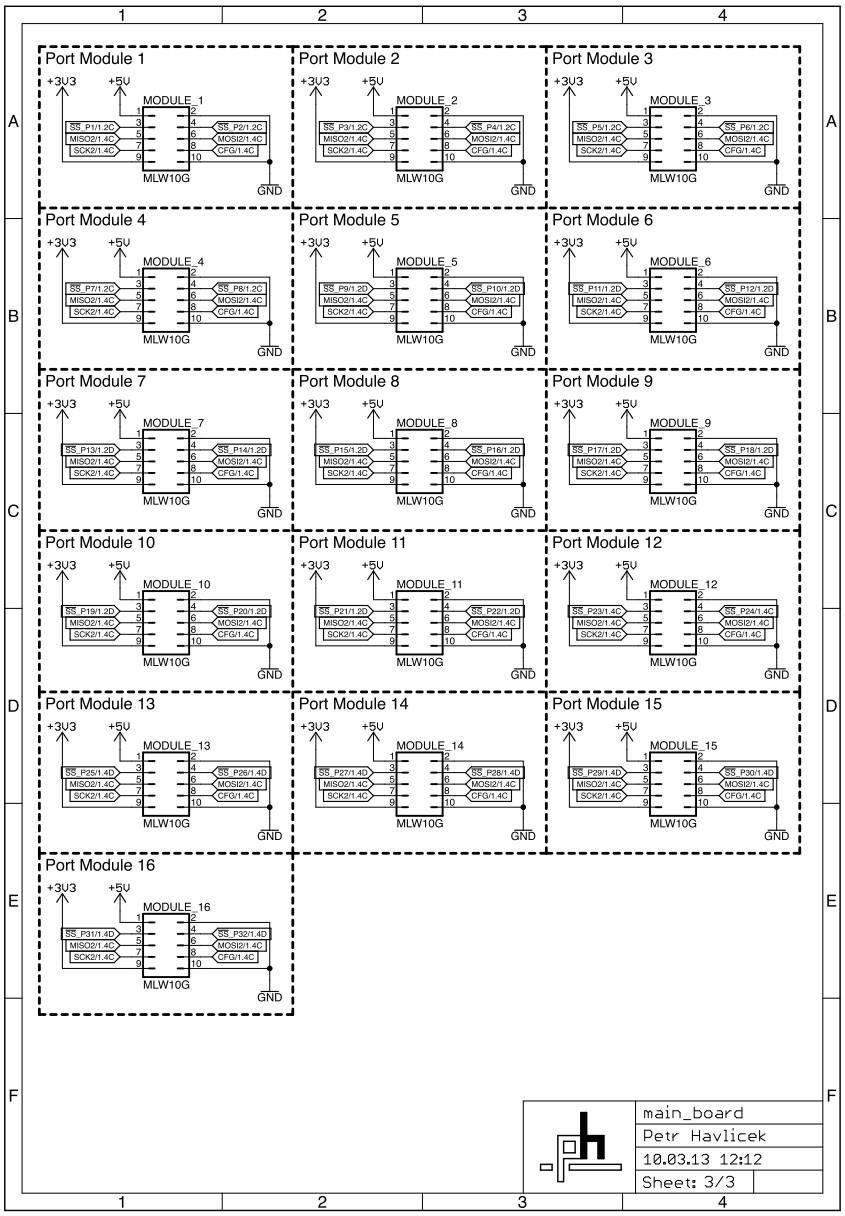
A Funkce ProcessPort

```
void ProcessPort(BYTE activePort, TCP_SOCKET portSocket) {
    BYTE totalPeriods = 0, toRecieve = 0, toSend = 0;
    BYTE *sendBuffer, *readBuffer;
    BYTE i, r = 0, w = 0;

    if (TCPIsConnected(portSocket))
        toSend = TCPIsGetReady(portSocket);

    SelectPort(activePort);

    WriteSPI2(toSend);              // Send number of bytes in input buffer
    if (ReadSPI2() != 21) {         // Check for connected port
        DeselectPort(activePort);
        return;
    }

    WriteSPI2(0);                   // Write dummy value
    toRecieve = ReadSPI2();         // Count of byte to receive
    WriteSPI2(0);                   // Write dummy value
    toSend = ReadSPI2();            // Free space in remote buffer

    // Allocate memory and get data from FIFO
    if (toSend) sendBuffer = (BYTE *)malloc(toSend);
    if (toRecieve) readBuffer = (BYTE *)malloc(toRecieve);

    if (TCPIsConnected(portSocket) && toSend)
        TCPGetArray(portSocket, sendBuffer, toSend);

    // Total number of transmit
    if (toSend > toRecieve) totalPeriods = toSend;
    else totalPeriods = toRecieve;
    if (totalPeriods > 40) totalPeriods = 40;

    // Data transfer
    for(i = 0; i < totalPeriods; i++) {
        if (w < toSend) WriteSPI2(sendBuffer[w++]);
        else WriteSPI2(0);
        if (r < toRecieve) readBuffer[r++] = ReadSPI2();
        else ReadSPI2();
    }

    DeselectPort(activePort);

    // Send data to socket
    if (TCPIsConnected(portSocket) && toRecieve)
        TCPPutArray(portSocket, readBuffer, toRecieve);

    // Free used memory
    if (toSend) free(sendBuffer);
    if (toRecieve) free(readBuffer);
}
```

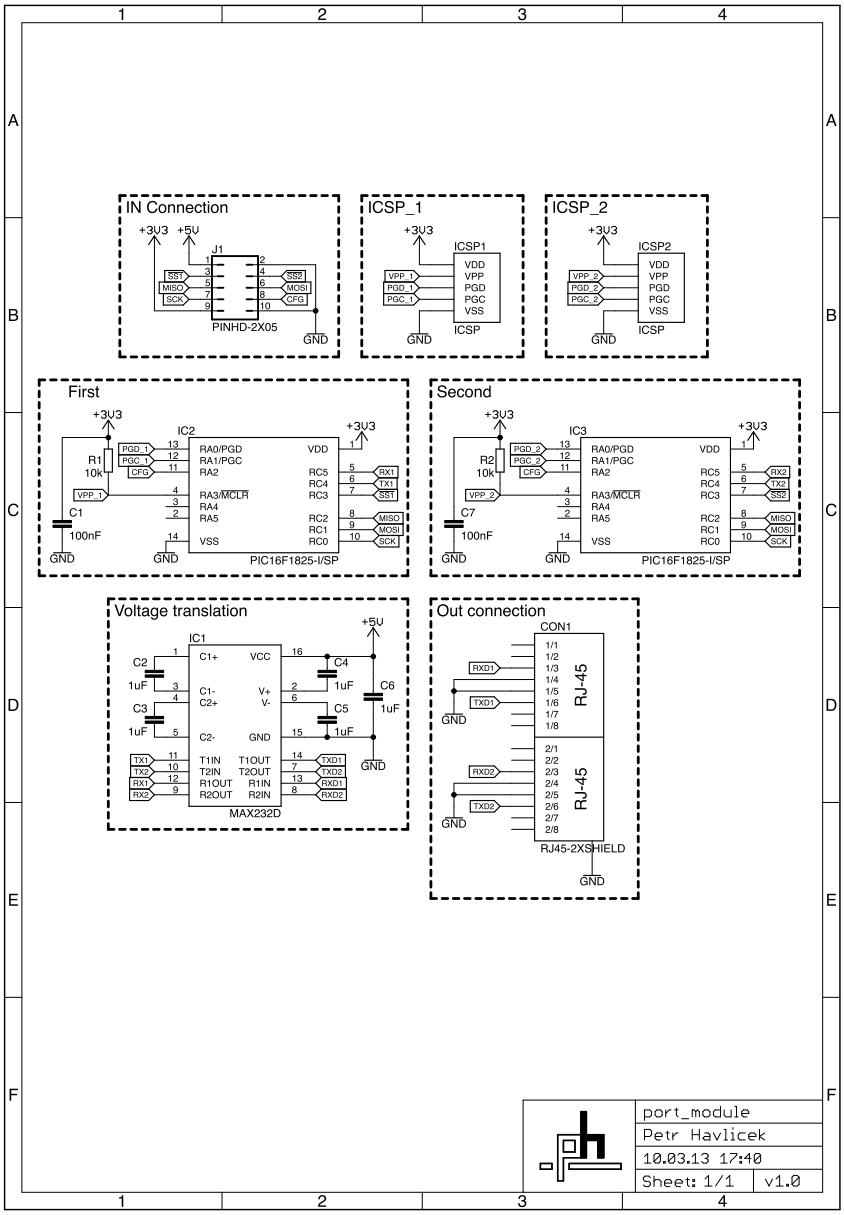
B Schéma hlavního modulu




USB
CON2
V+
D-
D+
GND
USB-B-TH
FB1 FBEAD
+3V3
IC3
VCC
VCCIO
USBDM
USBDP
RESET
OSCI
OSCO
3V3OUT
GND
GND
GND
GND
TEST
TXD
RXD
RTS
CTS
DTR
DSR
DCD
RI
CBUS0
CBUS1
CBUS2
CBUS3
CBUS4
U2RX/1.4D
U2TX/1.4D
C13
10nF
C14
100nF
C15
100nF
C16
4.7uF
R7
270R
R8
270R
LED3
TX
LED4
RX
Ethernet
FB2 FBEAD
CON3
MAG
TD+
CT_T
TD-
RD+
CT_R
RD-
GND
GND
RJ-45
J0026D21BNL
Green
Yellow
LEDA
LEDB
IC4
ENC28J60/SS
RESET
INT
SS
SCK
MOSI
MISO
OSC1
OSC2
CLKOUT
TPOUT+
TPOUT-
TPIN+
TPIN-
LEDA
LEDB
VCAP
RBIAS
VDD
VDDOSC
VDDPLL
VDDRX
VDDTX
VSS
VSSOSC
VSSPLL
VSSRX
VSSTX
ENC_RESET/1.4C
ENC_INT/1.2D
ENC_SS/1.4C
SCK1/1.4D
MOSI1/1.4C
MISO1/1.4C
R9
49.9
R10
49.9
R11
49.9
R12
49.9
C17
100nF
C18
100nF
R13 2k2
R14 2k2
Q2
25MHz
C19
22p
C20
22p
TP4
R15
2k32
C24
10uF
C22
100nF
C23
100nF
C26
100nF
EEPROM
IC5
EE_SS/1.4D
SCK1/1.4D
MOSI1/1.4D
MISO1/1.4C
SS
SCK
MOSI
MISO
VDD
VSS
25AA02E48
C25
100nF
main_board
Petr Havlicek
10.03.13 12:12
Sheet: 2/3


1
2
3
4
A
B
C
D
E
F
Port Module 1
MODULE_1
SS_P1/1.2C
SS_P2/1.2C
MISO2/1.4C
MOSI2/1.4C
SCK2/1.4C
CFG/1.4C
+3V3
+5V
MLW10G
GND
Port Module 2
MODULE_2
SS_P3/1.2C
SS_P4/1.2C
Port Module 3
MODULE_3
SS_P5/1.2C
SS_P6/1.2C
Port Module 4
MODULE_4
SS_P7/1.2C
SS_P8/1.2C
Port Module 5
MODULE_5
SS_P9/1.2D
SS_P10/1.2D
Port Module 6
MODULE_6
SS_P11/1.2D
SS_P12/1.2D
Port Module 7
MODULE_7
SS_P13/1.2D
SS_P14/1.2D
Port Module 8
MODULE_8
SS_P15/1.2D
SS_P16/1.2D
Port Module 9
MODULE_9
SS_P17/1.2D
SS_P18/1.2D
Port Module 10
MODULE_10
SS_P19/1.2D
SS_P20/1.2D
Port Module 11
MODULE_11
SS_P21/1.2D
SS_P22/1.2D
Port Module 12
MODULE_12
SS_P23/1.4C
SS_P24/1.4C
Port Module 13
MODULE_13
SS_P25/1.4D
SS_P26/1.4D
Port Module 14
MODULE_14
SS_P27/1.4D
SS_P28/1.4D
Port Module 15
MODULE_15
SS_P29/1.4D
SS_P30/1.4D
Port Module 16
MODULE_16
SS_P31/1.4D
SS_P32/1.4D
main_board
Petr Havlicek
10.03.13 12:12
Sheet: 3/3

C Schéma koncového modulu



D Obsah CD

- **Docs** - Dokumenty použité při vývoji zařízení.
  - **Datasheet** - Katalogové listy jednotlivých části systému.
  - **PIC32** - Dokumentace pro práci s PIC32.
  - **TCPIP** - Dokumenty popisující TCP/IP Stack.
- **Main_board** - Složka obsahující data ohledně hardwaru, schémata a návrhy desek.
  - **HW** - Projekt programu EAGLE s HW návrhem hlavního modulu.
  - **SW** - Projekt s Firmware pro hlavního modulu.
- **Port_module** - Složka s firmware pro jednotlivé moduly.
  - **HW** - Projekt programu EAGLE s HW návrhem koncového modulu.
  - **SW** - Projekt s Firmware pro koncový modul.
- **Klient_app** - Zdrojová aplikace napsaná v jazyce Perl.
- **Text** - Zdrojová data pro tvorbu tohoto dokumentu včetně obrázků.
  - **obr_zdroj** - Zdrojové soubory obrázků v programu Omnigraffle.